Titre: Title:	Multitraitement et processeurs configurables sur une plate-forme de haut niveau
Auteur: Author:	Bruno Lavigueur
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lavigueur, B. (2004). Multitraitement et processeurs configurables sur une plate-forme de haut niveau [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7406/

UNIVERSITÉ DE MONTRÉAL

MULTITRAITEMENT ET PROCESSEURS CONFIGURABLES SUR UNE PLATE-FORME DE HAUT NIVEAU

BRUNO LAVIGUEUR
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
NOVEMBRE 2004

Your file Votre référence
ISBN: 0-494-01351-6
Our file Notre référence
ISBN: 0-494-01351-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

MULTITRAITEMENT ET PROCESSEURS CONFIGURABLES SUR UNE PLATE-FORME DE HAUT NIVEAU

présenté par: LAVIGUEUR Bruno
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

Mme. NICOLESCU Gabriela, Doct., présidente
M. BOIS Guy, Ph.D., membre et directeur de recherche
M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche
M. SAVARIA Yvon, Ph.D., membre

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche Guy Bois pour le support qu'il m'a apporté ainsi que sa confiance et sa disponibilité. Je désire aussi souligner les remarques pertinentes sur le projet apportées par mon codirecteur Mostapha Aboulhamid. Il est aussi important de souligner l'aide financière que le NATEQ et le ReSMiQ m'ont fourni sous forme de bourses et qui m'a permis de me consacrer entièrement à ma recherche.

J'aimerais également remercier mes collègues qui m'on aidé de près ou de loin pour ce projet soit Luc Filion, Olivier Benny, Marc Bertola et François Deslauriers. Je tiens aussi à souligner le groupe Circus en entier pour son environnement de travail stimulant et pour de nombreux bons moments passés ensemble. Une mention spéciale va à David Quinn avec lequel j'ai travaillé étroitement pour la réalisation de ce travail. Son aide à été particulièrement utile au niveau de l'intégration et de l'utilisation de Click, des applications réseaux et des simulations.

Je tiens aussi à remercier l'équipe *SoC Platform Automation Group* de la compagnie ST Microelectronics à Ottawa pour leur don de StepNP ainsi que pour leur support technique et leurs nombreuses bonnes idées qui ont permis d'orienter cette recherche.

Je désire finalement remercier ma copine Claudia pour ses nombreux encouragements ainsi que la relecture et correction, parfois pénibles, de ce document.

RÉSUMÉ

Les systèmes embarqués occupent depuis bon nombre d'années une place importante sur le marché des semi-conducteurs. Avec les poussées technologiques récentes, ces systèmes embarqués sont souvent implémentés sur une puce unique et sont alors nommés « système-sur-puce ». Un système-sur-puce (SoC) permet d'atteindre un niveau de performance, de miniaturisation et de consommation de puissance difficile à obtenir avec les anciens types de circuits. Cependant, les systèmes-sur-puces sont complexes à développer et dispendieux à fabriquer. Afin de diminuer les coûts de développement, il est nécessaire d'augmenter le niveau d'abstraction lors de la spécification du système et aussi de maximiser la réutilisation.

Le développement récent des processeurs configurables offre une nouvelle solution pour s'attaquer au problème du temps de conception d'un SoC. Un processeur configurable peut être facilement modifié et étendu afin d'offrir des instructions spécialisées pour une classe d'application donnée. Dans biens des cas, l'emploi d'un processeur configurable génère des gains similaires à ceux atteignables avec un coprocesseur. Puisque les nouvelles instructions sont automatiquement intégrées dans le processeur configurable, il est plus facile à créer et à utiliser qu'un coprocesseur classique tout en offrant, dans bien des cas, des performances comparables. Le premier volet de cette recherche se penche sur l'utilisation des processeurs configurables dans la création d'un processeur réseau. Une méthodologie de conception et de partitionnement mieux adaptée aux processeurs configurables est proposée et employée. Cette méthodologie privilégie l'emploi d'instructions spécialisées devant l'utilisation de coprocesseurs afin d'obtenir rapidement un circuit ayant les performances désirées.

Pour réaliser nos expériences, une plate-forme permettant l'exploration architecturale à haut niveau, nommée StepNP, a été utilisée. StepNP est basé sur la bibliothèque SystemC 2.0 qui est aujourd'hui devenue un des langages les plus utilisés pour la des-

cription de systèmes hétérogènes à haut niveau. Le processeur Xtensa de Tensilica a été sélectionné de son côté comme processeur configurable et nous avons intégré son simulateur de jeu d'instructions dans StepNP. À l'intérieur de cet environnement, une plate-forme d'un processeur réseau et deux applications représentatives ont été développées. En appliquant notre méthodologie, il a été possible d'obtenir des accélérations significatives pour ces deux applications.

Le second volet de cette recherche porte sur l'utilisation d'un processeur capable de supporter plusieurs processus en parallèle et offrant des changements de contexte rapides. Il est démontré que la latence des communications peut avoir un impact très négatif sur les performances d'un processeur réseau et ce type de processeur semble être en mesure de minimiser les effets d'une latence élevée. Un modèle simulable d'un processeur supportant plusieurs fils d'exécution concurrents a donc été construit à partir du simulateur du Xtensa. Il est, par la suite, démontré que cette modification permet d'aller chercher des gains importants qui sont complémentaires à ceux offerts par les instructions spécialisées.

ABSTRACT

For a number of years, embedded systems have occupied a predominant place on the semi-conductor market. With all the recent technological advances, these embedded systems are more and more implemented on a single chip and are therefore called systems-on-a-chip, or SoC. A SoC enables us to attain a higher level of performance, better miniaturisation and lower power consumption than traditional multi-chip circuits. However, today's SoCs have become quite complex to develop and expensive to fabricate. In order to lower those development costs and be competitive on the market, it is now mandatory to raise the level of abstraction used to specify the system and also to maximize reuse between projects.

The recent advent of configurable processors offers a new solution to solve the problem of the long SoC development cycle. A configurable processor can be modified and extended in order to offer specialized instructions adapted to a certain application domain. Since the new instructions are automatically intergraded into the configurable processor, it is easier to create and use then the classical coprocessor solution, while at the same time offering an equivalent level of performance. The first part of this research looks at the possible uses of a configurable processor inside a network processor. A conception and partitioning methodology which is better adapted for configurable processors is proposed. This methodology advocates the usage of specialized instructions over coprocessors in order to quickly obtain a circuit with the desired level of performances.

In order to realize our experiments, a platform enabling high level and rapid architectural exploration has been used. This platform, named StepNP, is based on the SystemC 2.0 library which has become the new the facto standard to describe heterogeneous systems at a high level of abstraction. The Xtensa configurable processor from Tensilica was selected for this research and its instruction set simulator

was integrated into StepNP. In this environment, a platform of a network processor and two representative network applications were developed. With the help of our methodology, we were able to obtain appreciable accelerations for both applications.

The second part of this research looks at the usage of processors supporting multiple concurrent threads in hardware and therefore supporting rapid context switches. It is a demonstrated fact that high communication latencies can have disastrous effects on the performances of a network processor. A hardware multithreaded processor seems to be able to hide the effect of such latencies. A model of a hardware multithreaded processor has been developed with the Xtensa simulator as a starting point. Afterwards, our simulations have shown that this modification can enable us to greatly accelerate the application and that this new acceleration is complementary to the one offered by customized instructions.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES NOTATIONS ET DES SYMBOLES

AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
BCA	Bus Cycle Accurate
CMP	On-Chip Multiprocessor
CPI	Cycles Par Instruction
DES	Data Encryption Standard
DLX	Nom d'un processeur RISC très simple
DMA	Direct Memory Access
DSOC	Distributed System-On-Chip
DSP	Digital Signal Processor
ESP	Encapsulating Security Payload
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
HDL	Hardware Description Language
HMT	Hardware Multithreading
ILP	Instruction Level Parallelism.
IPsec	Internet Protocol for security
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
IPv4	Internet Protocol version 4
JTAG	Joint Test Action Group
LPDP	LISA Processor Design Platform
MIPS	Million d'instructions par seconde

MMU	Memory Management Unit
MP	Multi-processeurs
NoC	Network-on-a-Chip
NUMA	Non-Uniform Memory Access
OCP	Open Core Protocol
OSCI	Open SystemC Initiative
PCA	Pin Cycle Accurate
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPC	Remote Procedure Call
RTL	Register Transfer Level
RTOS	Real Time Operating System
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SoC	System-on-a-Chip
SOCGEN	SoC Generator
SOCMON	SoC Monitor
SOCP	SystemC OCP
SPD	Simple Packet Device
StepNP	System-level Exploration Platform for Network Processing
TCP/IP	Transmission Control Protocol / Internet Protocol
TF	Timed Functional
TIE	Tensilica Instruction Extension
TLM	Transaction Level Modeling
TLP	Thread Level Parallelism
TTL	Time To Live

UTF	Untimed functional
VCI	Virtual Component Interface
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Scale Integrated Circuit
VLIW	Very Long Instruction Word
VSIA	Virtual Socket Interface Alliance
XCC	Xtensa C Compiler
XLMI	Xtensa Local Memory Interface

LISTE DES ANNEXES

INTRODUCTION

Les systèmes embarqués, c'est-à-dire les systèmes dédiés à une tâche spécifique dans un environnement donné, occupent la part du lion sur le marché des semi-conducteurs. Avec les poussées technologiques des dernières années, ces systèmes embarqués sont souvent implémentés sur une puce unique et sont nommés « système-sur-puce » ou SoC (*system-on-chip*). L'utilisation de SoC dans des domaines aussi variés que les télécommunications, la vidéo, le traitement d'image, l'automatisation, l'industrie automobile ou le domaine médical s'est grandement accrue depuis la fin des années 1990. Ces circuits offrent aujourd'hui une multitude de fonctionnalités, sont complexes à développer et surtout dispendieux à fabriquer. Par exemple, avec un procédé de fabrication utilisant une technologie 90nm, le coût des masques excède le million de dollars et l'ensemble des coûts non récurrents peuvent atteindre les 100 millions [47]. Avec de telles dépenses, il est nécessaire de vendre une très grande quantité de circuits afin de récupérer son investissement initial. Cela explique en partie la popularité des circuits reprogrammables (comme un FPGA) pour les situations où le volume de distribution sera faible ou même moyen. Face à ces défis, l'industrie doit s'adapter et, pour ce faire, une approche souvent mise de l'avant est la réutilisation.

Outre le coût, un autre défi de taille auquel les créateurs de SoC doivent faire face est la compétition féroce entre les manufacturiers. Cela les pousse à réduire le plus possible le temps de développement d'un SoC afin d'être les premiers sur le marché. Il est aussi nécessaire de sortir de nouveaux produits avec de nouvelles fonctionnalités rapidement et souvent afin d'intéresser les consommateurs et de garder un avantage compétitif. L'importance de réutiliser non seulement des sous-modules d'un circuit, mais bien la majorité du design d'un Soc prend donc de l'ampleur. Pour ce faire, une approche de conception orientée plate-forme (*Platform-Based Design*) est utilisée [44]. Le terme plate-forme ne représente pas uniquement l'architecture du système (i.e. son organisation interne) en soi, mais plutôt une structure de base uti-

lisée par les ingénieurs pour construire le SoC et l'application qui s'exécutera dessus. L'utilisation de plates-formes architecturales constitue un premier moyen pour favoriser la réutilisation. Le second moyen est l'utilisation du logiciel pour implémenter la majorité des fonctionnalités du système embarqué. Les SoC sont de plus en plus composés d'un grand nombre de processeurs programmables, puisque le logiciel est plus facile à réaliser et à modifier que le matériel. Cela s'effectue au détriment des circuits matériels dédiés (ASIC) à l'intérieur du SoC, ces derniers demeurent néanmoins présents afin d'offrir aux processeurs certaines fonctions optimisées. De plus, pour pouvoir offrir une plus grande flexibilité, ces circuits peuvent posséder une partie de logique reconfigurable (comme un FPGA) dans lequel des blocs de logique plus spécialisés seraient implémentés.

Plusieurs compagnies ont déjà utilisé le concept d'une plate-forme réutilisable dans différents produits [15]. Notons entre autres, Phillips avec le Nexperia, Texas Instruments et le OMAP (*Open Multimedia Application Platform*) ou encore ST Microelectronics et le Nomadik. Cette solution permet de modéliser un SoC à partir d'une plate-forme d'exécution standard sur laquelle on insérera des blocs prédéfinis en fonction de l'application à concevoir. Lorsque la plate-forme implémente la majorité de ses fonctionnalités au niveau du logiciel, il est même possible de créer un nouveau produit sans modifier le circuit du SoC, ou dans d'autres cas en apportant des modifications mineures. L'investissement en temps de développement et en coûts de production peut alors être récupéré sur un plus grand nombre de circuits vendus. Afin de connaître du succès, cette approche demande cependant d'investir plus de ressources dans le développement d'un environnement de programmation flexible, puissant et surtout facile à utiliser qui permettra de tirer le maximum des capacités de la plate-forme.

Problématique

Lorsqu'une nouvelle plate-forme est développée, un grand nombre de problèmes sont soulevés, tant au niveau de la fabrication que de la conception. Un d'entre eux est d'effectuer une augmentation du niveau d'abstraction lors de la conception et de la simulation du modèle haut niveau qui agit alors comme spécification fonctionnelle. Cela s'avère essentiel pour être capable de tester plusieurs scénarios différents lors du design et permet d'éviter des retours en arrière extrêmement coûteux lorsque la création du SoC est déjà avancée. Ce design au niveau système permet non seulement de simuler et d'analyser l'architecture du SoC, mais aussi de s'occuper des interactions entre le logiciel et le matériel dès le début du cycle de développement. De plus, l'élévation du niveau d'abstraction s'avère nécessaire pour augmenter la productivité des concepteurs et permettre une pleine utilisation des ressources disponibles sur la puce. Par exemple, il est aujourd'hui possible d'intégrer des centaines de processeurs à l'intérieur d'une puce. Cependant, la complexité d'un tel circuit et l'absence d'outils de conception avancés rendent cette tâche presque impossible.

Pour attaquer ces problèmes, il est donc nécessaire de disposer d'outils de développement appropriés. La bibliothèque SystemC 2.0, qui est basée sur le langage C++, offre un bon point de départ et connaît une popularité grandissante en ce moment. SystemC offre différents niveaux d'abstraction et permet de créer un modèle fonctionnel d'un SoC. Cependant, SystemC ne propose pas de méthodologie de conception et offre un support limité pour le profilage du système ainsi que pour la simulation de la partie logicielle. C'est pourquoi son utilisation est habituellement combinée avec celle d'un autre outil qui vient combler ses lacunes. StepNP, qui sera décrit plus en détails au chapitre 2, est un de ces outils qui tente de faciliter le design d'un SoC. Il s'agit d'un outil de développement obtenu au travers d'une collaboration avec ST Microelectronics et il a été sélectionné pour créer la plate-forme simulable utilisée dans cette recherche.

Le domaine du traitement de paquets et des routeurs est une classe d'application qui demande des circuits performants et flexibles et qui constitue un bel exemple pour valider de nouvelles idées et méthodes de design au niveau système. Les routeurs traditionnels étaient construits autour d'un processeur général qui s'occupait de tout le traitement nécessaire. Avec le grand succès que connaît l'Internet et qui entraîne une augmentation de la demande de bande passante ainsi qu'avec l'arrivée de plusieurs nouveaux protocoles tels le transport de voix sur IP (*Internet Protocol*) ou encore le chiffrement des communications, ces routeurs traditionnels ne répondent plus à la demande. Un nouveau type de circuit, que l'on nomme NPU (*Network Processing Unit*), est apparu pour combler cette lacune. Un NPU est un système-sur-puce complexe qui permet la création de routeurs beaucoup plus performants en intégrant sur une même puce un ensemble de processeurs et coprocesseurs spécialisés, une structure de mémoire avancée ainsi que des interconnexions et des interfaces à haute vitesse. Un grand nombre de compagnies offrent déjà des circuits de ce type sur le marché [75] et les différents NPU commerciaux utilisent parfois des architectures radicalement différentes pour arriver à des résultats semblables. Cela illustre bien à quel point la quantité de solutions possibles est grande lorsque vient le temps de créer un SoC et pourquoi il est nécessaire d'explorer rapidement plusieurs architectures différentes avant de se lancer dans la création de l'une d'entre elles. Finalement, une dernière problématique soulevée par les NPU est l'intégration du logiciel avec les composants matériels de la plate-forme. Cet aspect devient de plus en plus important et, dans certains cas, l'effort de développement du logiciel est même rendu supérieur à celui nécessaire pour le matériel [47]. Cela illustre bien à quel point il est important de disposer de moyens efficaces pour programmer un NPU.

Objectifs

L'objectif principal de ce travail consiste à évaluer, à haut niveau, les avantages d'utiliser des processeurs configurables dans la conception des processeurs réseaux. Les NPU possèdent typiquement un grand nombre de processeurs embarqués avec des instructions spécialisées et des accélérateurs matériels. Les processeurs configurables, qui permettent de rapidement générer et intégrer des instructions personnalisées et qui offrent aussi des outils de développement logiciels automatiquement générés, semblent être en mesure d'offrir des gains au niveau de la performance et du temps de développement [32]. L'objectif ici n'est pas de créer un nouveau processeur réseau complet capable de rivaliser avec les NPU commerciaux, mais bien d'évaluer les impacts que les processeurs configurables peuvent avoir à l'aide d'un outil d'exploration rapide. Nous visons néanmoins à obtenir un processeur réseau flexible et facilement programmable qui est capable de fournir des performances intéressantes. Le deuxième objectif est de se pencher sur les techniques de modélisation d'un processeur capable de traitement multiprocessus ainsi que sur les avantages qu'un tel processeur peut apporter dans la conception d'un NPU, particulièrement au niveau de la latence des communications à l'intérieur de la puce.

Méthodologie

Afin de répondre aux objectifs, il est d'abord nécessaire de se familiariser avec la plate-forme de développement StepNP et les fonctionnalités qu'elle offre. C'est avec StepNP et SystemC que la plate-forme du NPU sera développée. Par la suite, il est nécessaire d'ajouter un modèle de processeur configurable dans StepNP puisqu'à la base il n'y en a pas de disponible. Pour cette recherche, notre choix s'est arrêté sur le Xtensa de Tensilica [25], car il s'agit d'une technologie mature qui permet de facilement ajouter des instructions au processeur. Comme troisième étape, le logiciel

de développement de routeur Click [45] est adapté afin de pouvoir fonctionner sur le Xtensa et interagir avec le reste de la plate-forme. Click est un environnement de développement de logiciels qui permet de rapidement créer un routeur complet. Deux applications réseaux développées avec Click serviront de banc d'essai pour vérifier l'impact des instructions personalisées. Finalement, un modèle simulable d'un processeur ayant l'architecture d'un Xtensa et supportant plusieurs processus concurrent sera développé et analysé.

Contribution

Bien qu'une multitude de tâches spécifiques ont dû être réalisées pour atteindre les objectifs de ce projet, une des principales contributions provient des modifications proposées à la méthodologie de co-design classique. Cette méthodologie, présentée dans [69], ajoute quelques étapes à l'approche traditionnelle et permet de tirer profit des avantages offerts par un processeur configurable ainsi qu'un environnement de développement à haut niveau. La seconde contribution est la réalisation d'un modèle simulable d'un processeur avec traitement multiprocessus qui, bien que le circuit physique correspondant n'existe pas, permet d'avoir une bonne idée des bénéfices offerts par une telle technologie dans un processeur réseau.

Distribution des chapitres

Le premier des quatre chapitres du mémoire compare certaines technologies permettant de créer un processeur configurable ainsi que des approches utilisées pour extraire du parallélisme d'une application et masquer la latence de certaines opérations grâce au traitement multiprocessus. Ce chapitre dresse aussi un bref tableau des processeurs réseau existants. Le second chapitre décrit les principaux outils utilisés pour cette recherche (StepNP, Click et le Xtensa) ainsi que les techniques pour intégrer un

simulateur de processeur comme le Xtensa dans StepNP. De plus, ce chapitre présente la méthodologie de codesign employée. Le troisième chapitre se penche, quant à lui, sur l'optimisation du jeu d'instructions pour le processeur réseau et montre les gains obtenus avec ces instructions spécialisées. Finalement, l'utilisation d'un modèle de processeur avec support pour plusieurs processus concurrents et les avantages de ce type de processeur seront présentés dans le chapitre 4.

CHAPITRE 1

REVUE DES PROCESSEURS CONFIGURABLES ET DES TECHNIQUES D'EXPLOITATION DU PARALLÉLISME

Depuis bon nombre d'années, il existe des processeurs spécialisés pour une certaine classe d'application qui sont nommés ASIP (*Application Specific Instruction set Processor*). Ces processeurs proposent un compromis intéressant entre la flexibilité offerte par les langages de programmation haut niveau et la performance offerte par ses unités de calcul et son architecture spécialisée. Développer un nouvel ASIP demande cependant beaucoup de temps et d'argent. Avec la complexité déjà présente pour le développement d'un SoC et les fenêtres de mise en marché petites, ce surcoût n'est pas le bienvenu et joue contre les ASIP en faveur de processeurs généreaux déjà existants. Cependant, depuis quelques années, il existe des moyens d'automatiser la création d'un ASIP ce qui rend son utilisation beaucoup plus intéressante. Le présent chapitre débute donc par une revue des différentes technologies qui ont comme objectif de permettre la création (ou la modélisation) rapide d'un ASIP et son intégration dans un circuit monopuce.

La seconde partie de cette revue se penche sur un deuxième facteur important pour atteindre des hautes performances dans un SoC : tirer avantage du parallélisme de l'application. Différentes approches pour exploiter le parallélisme et masquer la latence des opérations plus coûteuses sont étudiées.

Finalement, les processeurs réseaux sont brièvement présentés puisqu'il s'agit d'une classe d'applications qui peut bénéficier grandement de l'utilisation de processeurs configurables et de processeurs capables de traiter plusieurs processus en parallèle.

1.1 Processeurs configurables

Tel qu'indiqué dans [19, 73], le domaine des processeurs configurables s'est développé rapidement au cours des dernières années. Dans la présente section, différentes technologies permettant de créer ou de modéliser des ASIP sont décrites et comparées, en commencent d'abord par la technologie qui a été utilisée pour cette recherche : le Xtensa.

1.1.1 Le processeur Xtensa

Le Xtensa de la compagnie Tensilica est un processeur configurable visant le marché des processeurs synthétisés à l'intérieur d'un système sur une puce (souvent appelé *processeur soft core*). Le processeur est configurable et extensible, c'est-à-dire que plusieurs options architecturales peuvent êtres sélectionnées ou non et aussi que l'utilisateur peut ajouter des nouvelles instructions au processeur afin d'étendre ses fonctionnalités [25, 88].

1.1.1.1 Un processeur RISC de base

Le Xtensa n'offre pas une liberté totale face aux caractéristiques du processeur qu'il est possible d'obtenir lors de la configuration puisqu'un ensemble d'éléments de base se retrouvent obligatoirement dans chaque configuration. Il s'agit tout d'abord d'un processeur RISC 32 bits possédant à la base un pipeline de 5 étages similaire à celui du DLX [33] c'est-à-dire qu'il possède les étages suivants, tel qu'illustré à la figure 1.1 :

- lecture de l'instruction (*load instruction*) ;
- décodage de l'instruction et accès aux registres impliqués (*decode/register fetch*) ;
- exécution de l'instruction (calculs des adresses pour les chargements et rangements,

opérations de l'UAL ou encore du multiplieur si présent, calcul de l'adresse de la prochaine instruction) ;

- accès mémoire (*memory access*) ;
- écriture des résultats et mise à jour des registres (*write back*).

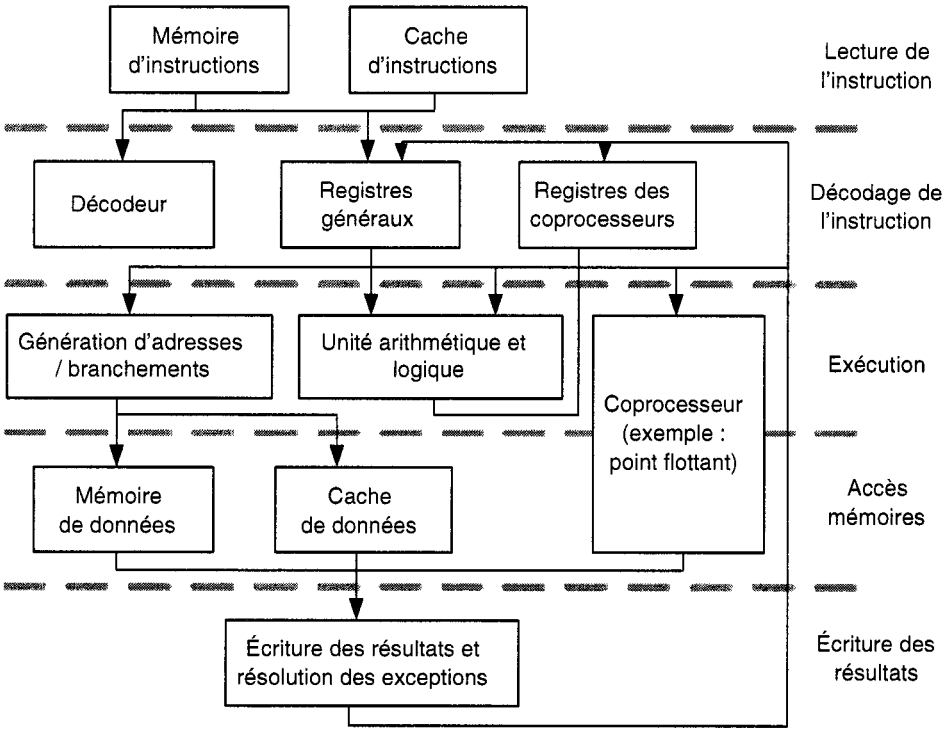


FIGURE 1.1 Pipeline du processeur Xtensa

Afin de réduire la taille du code une fois compilé, les instructions du Xtensa sont encodées sur 24 bits et les instructions les plus fréquemment utilisées ont aussi une représentation sur 16 bits. Certains processeurs embarqués, tel le ARM7, possèdent des instructions sur 32 ou 16 bits et utilisent un bit d'état interne afin de savoir quel type d'instruction est présentement lu de la mémoire. Lorsque le type d'instruction traité change, le bit d'état du processeur doit être explicitement modifié par le code de l'application. Dans le cas du Xtensa les instructions de 16 et 24 bits peuvent être librement entremêlées puisque l'encodage de l'instruction indique sa taille. Le Xtensa est donc capable de lire la mémoire d'instruction 32 bits à la fois et de décoder les instructions (dans le premier étage du pipeline) sans changer d'état.

Le jeu d'instructions du processeur, couramment appelé *Instruction Set Architecture* (ISA), comprend 80 instructions RISC qui englobent les instructions classiques d'une architecture RISC tel le DLX [33] et quelques instructions de plus comme un support pour les boucles rapides (le branchement, l'incrémentation du compteur et la condition arrêt sont calculés simultanément).

Finalement le Xtensa utilise un mécanisme de fenêtres coulissantes (*sliding windows*) semblable à celui utilisé dans les processeurs Sparc de Sun ou encore le NIOS de Altera pour gérer ses appels de fonctions. Le processeur dispose d'un plus grand nombre de registres physiques que de registres logiques visibles par le programmeur. Lors d'un appel de fonction, la fenêtre se déplace afin d'offrir un nouvel ensemble de registres au programme et lorsque la fonction est terminée, la fenêtre revient à sa position initiale ce qui permet au processeur de continuer l'exécution de la fonction mère. Le registre de pointeur de pile (*stack pointer*) et les autres registres d'états se trouvent donc sauvegardés dans la fenêtre coulissante. Il y a une superposition entre les deux fenêtres afin de permettre l'envoi de paramètres et le retour des réponses entre les fonctions mères et filles. Les fenêtres coulissantes ont comme principal avantage d'accélérer les appels de fonctions puisqu'il devient inutile de sauvegarder l'état des registres dans une pile avant de sauter dans la sous-routine.

1.1.1.2 Un processeur configurable

Mis à part les caractéristiques décrites ci-dessus, le Xtensa offre plusieurs options de configurations à l'utilisateur qui le démarque des processeurs RISC de base. En effet, à travers un outil de configuration graphique disponible sur le site Internet de Tensilica[1], l'usager peut effectuer plusieurs choix architecturaux qui vont modifier le processeur. Tout d'abord il est possible d'ajouter certains blocs logiques au processeur de base afin d'en augmenter les capacités, plus précisément il est possible d'inclure :

[1]http ://www.tensilica.com

- une unité de multiplication/accumulation (MAC) ;
- un multiplicateur en virgule fixe de 32 bits ;
- un coprocesseur point flottant ;
- des compteurs servant d'horloges (*timer*), utilisables par exemple par un système d'exploration temps réel (RTOS) ;
- de la circuiterie pour le déverminage du circuit physique (JTAG) ;
- une série d'extensions nommées Vectra qui offrent des capacités DSP (avec entre autre des instructions SIMD) [21] ;
- une unité de gestion de la mémoire (MMU) permettant de protéger des zones en cas d'exception sur le processeur qui rendraient ces zones temporairement invalides et aussi d'effectuer une conversion entre les adresses physiques et virtuelles ;
- un port d'interface supplémentaire, nommée *Xtensa Local Memory Interface* (XLMI), pour brancher une mémoire locale rapide (ou n'importe quel périphérique adressable comme une mémoire).

Par la suite, il est possible de faire varier certains paramètres architecturaux du processeur. Par exemple la largeur de l'interface du Xtensa avec la mémoire peut varier de 32 à 128 bits, ou encore le nombre de registres disponibles pour la fenêtre coulissante peut être modifié. Le tableau 1.1 indique certains de ces paramètres modifiables ainsi que les valeurs possibles.

Une fois toutes les options sélectionnées, les choix sont validés afin de s'assurer que la configuration ne comporte pas d'erreurs, puis le processeur désiré est généré. Par la suite, le code RTL, les outils de développement logiciel et les simulateurs peuvent être téléchargés à même le site Internet où la configuration a eu lieu. Finalement, notons que l'interface de configuration offre aussi des estimés de la taille, la puissance et la vitesse du processeur [72]. Ces estimés sont mis à jour à toutes les fois qu'une option de la configuration est modifiée.

TABLEAU 1.1 Paramètres configurables sur le Xtensa

Paramètre	Valeurs possibles
Taille des caches (instructions et données)	0, 1, 2, 4, 8, 16, 32 KOctets
Type des caches	À correspondance directe ou associative par 2, 3 ou 4. Écriture simultanée ou réécriture
Largeur d'une ligne de la mémoire chache	16, 32, 64 Octets
Taille de la fenêtre coulissante	32 ou 64 registres
Taille du bus externe	32, 64 ou 128 bits
Nombre d'interruptions	0 à 32
Niveau de priorité des interruptions	0 à 6
Nombre de compteurs (timers)	0 à 3
Ordonancement de la mémoire	Big-endian ou little-endian

1.1.1.3 Un processeur extensible

La force principale du Xtensa ne se situe pas tant au niveau des options qui peuvent être choisies au moment de la configuration, qu'au niveau de la possibilité d'étendre soi-même la fonctionnalité du processeur en ajoutant des instructions spécialisées à l'ISA. Les nouvelles instructions sont décrites à l'aide d'un langage nommé *Tensilica Instruction Extension* (TIE). Le langage TIE permet de décrire l'encodage de nouvelles instructions ainsi que leur nom mnémonique, les opérations arithmétiques effectuées et les registres utilisés. Il est relativement simple de décrire en TIE une instruction qui effectue des manipulations de bits non standard ou encore des opérations arithmétiques particulières. TIE offre aussi la possibilité de créer des nouveaux registres internes au processeur avec la taille désirée. La syntaxe du langage TIE s'inspire directement de celle du Verilog et elle est donc bien adaptée à la description du matériel. Il est simple d'écrire des instructions TIE, car les détails de l'implémentation sont complètement abstraits laissant à l'usager uniquement la tâche de décrire la logique combinatoire à effectuer. Toute la logique nécessaire à l'ajout de nouvelles

instructions dans le pipeline du processeur et à la gestion des items tel le décodage de l'instruction, l'insertion de bulles dans le pipeline, les inter-blocages et les chemins d'envois est automatiquement générée à partir de la description TIE. La figure 1.2 illustre de manière simplifiée la logique ajoutée au pipeline pour une instruction TIE. Bien que le code RTL du processeur généré soit disponible, il n'est donc pas nécessaire de le modifier manuellement afin d'étendre les fonctionnalités du Xtensa.

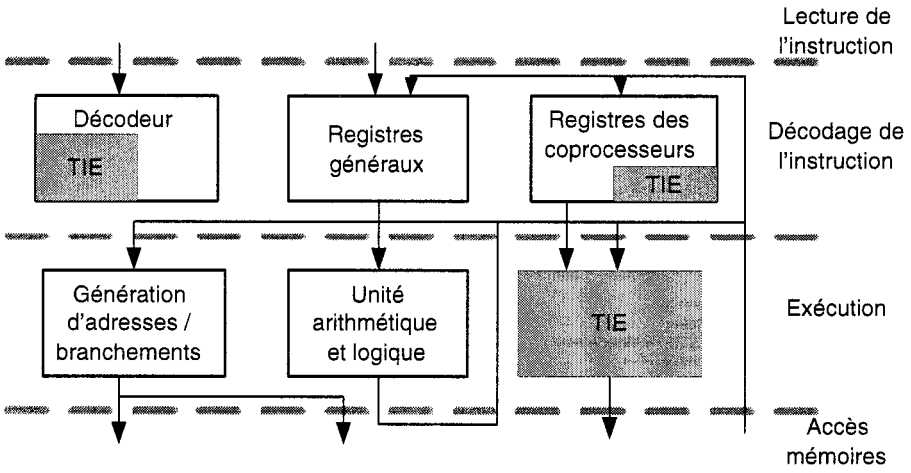


FIGURE 1.2 Ajout d'instructions TIE dans le pipeline

1.1.1.4 Les outils logiciels fournis avec le Xtensa

Afin de profiter des fonctionnalités offertes par le Xtensa, une suite d'outils logiciels est offerte. Elle comprend entre autre :

- un compilateur C et C++ de GNU ainsi que la suite standard d'outils de compilation (éditeur de liens, assembleur, etc...) ;
- un compilateur fourni par Tensilica (*Xtensa C Compiler* : XCC) ;
- un dévermineur (GDB et Xray de Mentor Graphics) ;
- un outil de profilage ;
- un simulateur de jeu d'instructions (ISS) ;
- un modèle simulable à l'intérieur de Seamless CVE de Mentor Graphics.

Ces outils sont capables de s'adapter aux différentes configurations du Xtensa. Cela s'avère particulièrement utile pour deux raisons. Tout d'abord, au niveau du compilateur C et C++, il est possible d'insérer facilement les nouvelles instructions TIE à l'intérieur du code de haut niveau, sans avoir besoin de passer par du code écrit en langage machine (la section 3.4 contient un exemple). Deuxièmement, il n'est pas nécessaire de modifier les outils de compilation et de simulation pour chaque nouvelle configuration, ce qui sauve énormément de temps de développement.

1.1.1.5 Limitations et méthodologie

Afin de mieux comprendre ce que le processeur Xtensa est et que ce qu'il peut faire, il est utile de décrire ce qu'il n'est pas ainsi que ses limitations. Tout d'abord, le Xtensa n'est pas un processeur reconfigurable dynamiquement. Une fois une configuration terminée, un processeur correspondant est généré et par la suite le code RTL résultant peut être intégré dans un design de système-sur-puce. À l'opposé, les processeurs reconfigurables comportent une certaine partie de logique matérielle reconfigurable qui peut être modifiée en cours d'exécution. Le désavantage principal de cette technique est le temps nécessaire pour reprogrammer le circuit logique sur la puce. Les processeurs reconfigurables dynamiquement en sont encore surtout au stade de la recherche, mais certaines solutions commerciales sont à l'horizon (voir l'annexe I.3).

L'extraction automatique d'instructions spécialisées est une autre caractéristique qui fait l'objet de recherches [4] et qui n'est pas incluse dans le Xtensa. Avec un outil d'extraction automatique, le code source est analysé afin d'identifier automatiquement les parties qui consomment beaucoup de temps de calcul. Par la suite, des instructions spécialisées sont générées et peuvent aussi être automatiquement insérées dans le code. La méthodologie proposée avec le Xtensa repose plutôt sur l'expérience des développeurs et consiste en les étapes suivantes :

1. Faire un profilage du code et déterminer soi-même les boucles à optimiser.
2. Isoler les parties à optimiser et puis créer une ou des instructions spécialisées pour effectuer la tâche.
3. Insérer manuellement les nouvelles instructions dans le code source C ou C++.
4. Réaliser un second profilage afin de vérifier si les résultats sont satisfaisants.
5. Itérer jusqu'à l'obtention des résultats voulus.

Bien entendu, rien n'empêcherait d'avoir un outil d'extraction d'instructions automatisé et d'insérer son utilisation dans la méthodologie décrite ci-dessus.

1.1.2 LISATek

LISATek est un outil permettant d'automatiser le design d'un processeur embarqué. LISATek, aujourd'hui devenu un produit commercial de la compagnie Coware, fut à l'origine un projet de recherche développé à l'université de Aachen en Allemagne. L'idée de base de LISATek est de permettre de décrire un modèle d'un processeur à partir du langage LISA (*Language for Instruction Set Architecture*). Puis, grâce à cette description, le processeur peut être généré, analysé, simulé et deverminé avec les outils fournis dans l'environnement LISATek.

1.1.2.1 Le langage LISA

LISA se veut être un langage de description de matériel (*Hardware description language*, HDL) qui, contrairement au VHDL ou au Verilog, se spécialise dans la description de processeurs embarqués. LISA comprend en effet des constructions qui permettent de décrire le processeur à un plus haut niveau d'abstraction qu'un langage HDL classique, par exemple il supporte une description rapide d'un pipeline ou d'une mémoire cache [65]. En ce sens LISA permet une description intermédiaire entre un modèle matériel très détaillé et un modèle simulable (ISS) à haut niveau. De plus,

contrairement à une description en VHDL, une description LISA permet d'extraire automatiquement plusieurs informations utiles, tels les détails du jeu d'instructions, afin de générer un compilateur et un ISS automatiquement. LISA vise remplir les objectifs suivants en tant que langage de description [92] :

- fournir un modèle précis au niveau des cycles d'horloge (*cycle accurate*) ;
- supporter une vaste gamme de classes de processeurs en passant des micro-contrôleurs simples aux processeurs RISC et aux machines plus complexes tel les processeurs vectoriels et les processeurs avec pipeline superscalaire ;
- offrir différents niveaux d'abstractions pour la modélisation du processeur. Par exemple, un processeur peut en premier lieu être décrit avec un modèle comportemental (behavioral) et simulé avant d'être raffiné vers un modèle architectural ;
- permettre de générer automatiquement les outils logiciels pour le processeur à partir de sa description LISA.

LISA possède une syntaxe inspirée du langage C et chaque description d'un processeur à l'aide de LISA est séparée en deux parties. Une première section nommée «ressource» permet de déclarer les mémoires, registres et pipelines disponibles au processeur, en d'autres mots, tous les éléments qui permettent de décrire l'état du processeur et qui servent de ressources pour les instructions. La seconde section nommée «opération» s'occupe de décrire la structure, le comportement et le jeu d'instructions du processeur. Il y a plusieurs blocs opérations dans la description d'un processeur, chacun ayant une certaine responsabilité propre, par exemple décrire une instruction ou encore le fonctionnement et le mode d'accès d'une banque de registres généraux (préalablement déclarés dans les ressources). Chaque bloc d'opération contient plusieurs sous sections dont :

- la section «*Coding*» qui définit l'encodage de l'instruction et les différents champs qui contiennent les opérandes et les paramètres ;
- la section «*Syntax*» qui définit la représentation mnémonique de l'instruction et de ses opérandes ;
- la section «*Behavior*» qui permet de décrire en C ou C++ le code comportemental

que l'instruction réalise, cette description est utilisée par le simulateur ;

- la section «*Activation*» qui indique si l'exécution d'une opération en déclenche une autre (par exemple, une addition déclencherait l'opération d'écriture du résultat dans un registre au cycle suivant) ;
- la section «*Declare*» qui stipule les noms d'opérandes locaux, comme par exemple le nom du champ de l'instruction qui indique le numéro du registre de sortie.

Le langage LISA supporte la déclaration de plusieurs pipelines dans un même processeur et supporte aussi des opérations classiques sur un pipeline tels le blocage, les hasards (structurel, de donnée et de contrôle), vider le pipeline en cas de hasard et des chemins d'envois. Notons aussi que LISA supporte des processeurs avec des instructions SIMD (*Single Instruction Multiple Data*) et VLIW (*Very Long Instruction Word*) ainsi que des pipelines superscalaires et l'exécution des instructions dans le désordre. Finalement, une opération peut être attribuée à un étage précis d'un pipeline dans le code LISA. Par exemple, lors de la description d'une opération qui effectue une addition, en plus de définir l'encodage de l'instruction et les ressources utilisées, il est aussi possible de spécifier à quel étage du pipeline l'addition aura lieu en tant que telle.

1.1.2.2 L'environnement LISATek

L'environnement de travail LISATek permet de générer automatiquement les outils de développement logiciel et le code RTL du processeur décrit avec le langage LISA, il permet aussi de simuler et de déverminer le processeur. Afin d'effectuer la génération automatique des outils de développement, différents modèles du processeur, tous décrits dans le code LISA, sont extraits et utilisés [34, 74].

- Le modèle des mémoires énumère les registres et les adresses mémoire accessibles, il spécifie aussi les tailles, les plages et les noms symboliques des mémoires. Ce modèle est utilisé, entre autre, par le compilateur lors de l'assignation des registres et aussi par le devermineur afin d'afficher l'état du processeur. Ce modèle est extrait de la

section «ressource» du code LISA.

- Le modèle des ressources décrit les ressources matérielles disponibles (mémoires, registres, pipelines) et il indique quelle ressource est utilisée par chacune des opérations du processeur ainsi que le moment de l'utilisation (l'étage du pipeline). Ce modèle permet donc au compilateur et au simulateur de vérifier l'ordonnancement des opérations.
- Défini à l'intérieur des opérations, le modèle du jeu d'instructions peut être extrait des sections «coding», «semantic» et «syntax». Ce modèle permet de décrire la syntaxe de l'assembleur ainsi que l'encodage et le mode d'adressage des instructions.
- Le modèle comportemental permet d'abstraire les fonctions réalisées par les unités fonctionnelles du processeur. Le code C contenu dans ce modèle est exécuté tel quel par l'ISS.
- Le modèle temporel spécifie la latence des instructions ainsi que les séquences d'activation des opérations.
- Le modèle architectural du processeur permet de regrouper différentes opérations (telles une addition et une soustraction) ensemble afin qu'elles soient réalisées par la même unité fonctionnelle matérielle.

À partir de ces différents modèles, ou vues, du processeur que l'on retrouve dans le code LISA, une chaîne complète d'outils logiciels peut être générée automatiquement. C'est l'environnement LISATek, autrefois nommé *LISA Processor Design Platform* (LPDP), qui se charge de générer non seulement les outils logiciels, mais aussi le simulateur et le code RTL du processeur. Dans les outils générés nous retrouvons [34] :

- un compilateur C/C++ ;
- un assembleur ;
- un programme d'édition de liens (*linker*) ;
- un dévermineur avec interface graphique ;
- un simulateur d'instructions (ISS) et une interface pour la co-simulation ;

- le code RTL en langage VHDL du chemin de contrôle du processeur.

Puisque l'environnement de développement (compilateur et simulateur) est automatiquement généré à partir de la description LISA, il est possible de garantir qu'il n'y a pas d'inconsistance entre le compilateur, l'ISS et les autres outils. De plus, il n'y a qu'un seul code à maintenir et à modifier afin d'effectuer des changements sur le processeur.

L'ISS généré par LISATek est capable de simuler les aléas du pipeline [66]. De plus, l'ISS supporte la simulation compilée [64]. Cette technique permet d'analyser le code compilé afin d'en accélérer la simulation, par exemple en décodant d'avance certaines instructions fréquentes ou encore en ordonnançant d'avance les instructions pour le pipeline. L'ISS est aussi fourni avec un API permettant de l'interfacer dans un environnement en SystemC simulant un système-sur-puce complet.

Le code VHDL est généré automatiquement pour le chemin de contrôle et les blocs de décodage d'instruction du processeur et de ses pipelines. Le contrôleur et les chemins d'envoi du pipeline sont aussi générés. Cependant, le code RTL n'est pas automatiquement prêt à être synthétisé puisque LISATek génère uniquement l'interface des unités fonctionnelles du chemin de données. Ces unités correspondent à la section comportementale des blocs opérations. Il est donc nécessaire d'écrire une partie du code RTL du chemin de données soi-même.

Finalement, notons que le compilateur de code C généré par LISATek n'est, pour l'instant, pas très performant et requiert encore l'écriture de certaines boucles en assembler afin de tirer avantage des instructions plus spécialisées de l'ISA. Cependant, la nouvelle version 2004.1 de LISATek utilise l'outil CoSy de la compagnie ACE [1] afin de générer le compilateur C et de meilleures performances sont annoncées.

1.1.2.3 Conclusions sur LISATek

LISATek est donc une plate-forme qui offre un haut niveau de flexibilité et d'automatisation. Avec cet outil, il est possible de rapidement développer un processeur embarqué avec des caractéristiques propres à une application ou une classe d'applications données. Tel que discuté dans [74, 34], les performances en terme de vitesse, de taille et de consommation de puissance ne sont pas aussi bonnes avec un processeur généré par LISATek qu'avec un processeur réalisé manuellement et le code VHDL généré par LISATek peut même nécessiter certaines retouches manuelles avant d'être prêt pour la production. Il reste néanmoins que LISATek offre une méthodologie simple et flexible qui permet de rapidement créer un ASIP à partir d'une spécification, tel qu'illustré à la figure 1.3. Si nous comparons la méthodologie de design LISATek avec la méthodologie de Tensilica, la principale différence qui en ressort est que LISATek offre une plus grande flexibilité au niveau du design du processeur, avec comme coût un sacrifice au niveau de la performance et du temps de développement.

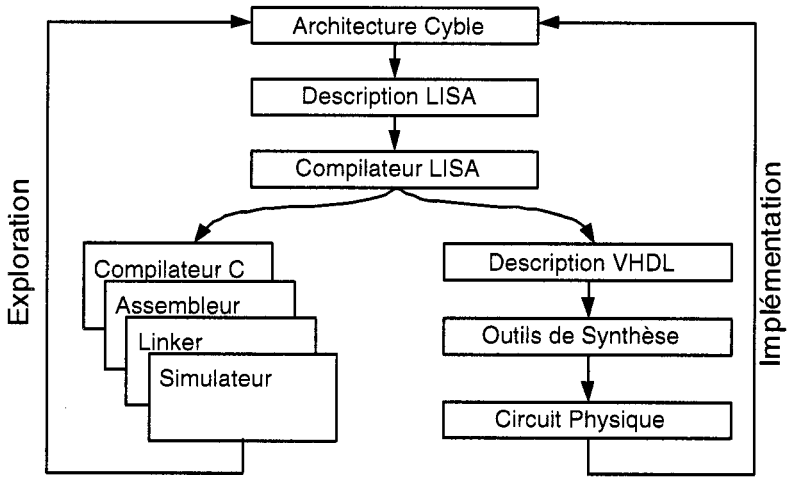


FIGURE 1.3 Flot de conception avec LISATek

1.1.3 SimpleScalar

SimpleScalar est un outil de simulation de processeurs à l'origine développé à l'université du Wisconsin et à l'université du Michigan et aujourd'hui supporté et développé par la compagnie SimpleScalar LCC[2]. L'outil est disponible gratuitement pour les universités ainsi que les groupes de recherche à but non lucratif et offre l'infrastructure nécessaire pour la création d'ISS spécialisés. De plus, le code source de SimpleScalar est aussi disponible gratuitement. Il en résulte que SimpleScalar est le simulateur le plus souvent employé pour valider des nouvelles idées dans les publications académiques portant sur les microprocesseurs.

SimpleScalar vise à atteindre trois objectifs principaux pour les modèles de processeurs : la performance de la simulation, la flexibilité du simulateur (si l'on veut modifier des détails du processeur) et le niveau de détails de la simulation. En pratique il est difficile d'optimiser ces trois critères en même temps. C'est pourquoi SimpleScalar offre dans sa version actuelle (v3.0) sept simulateurs opérant à des niveaux d'abstractions différents. Le tableau 1.2, provenant de [6], indique les performances relatives, en million d'instructions exécutées par secondes (MIPS), de chacun de ces simulateurs.

À la base, l'ISA supporté par SimpleScalar est semblable à celle d'un processeur MIPS[3] IV [67]. Les vitesses de simulation du tableau 1.2 sont toutes obtenues avec ce jeu d'instructions. Cet ISA particulier est nommé PISA (*Portable Instruction Set Architecture*) par les développeurs de SimpleScalar et toutes les instructions sont encodées sur 64 bits afin de faciliter l'ajout de nouvelles instructions. À moins de recréer un nouveau modèle de processeur, cela limite donc l'exploration du jeu d'instructions à des raffinements de celui du MIPS. En ce sens, il est aussi possible

[2]http ://www.simplescalar.com

[3]MIPS est le nom du processeur, à ne pas confondre avec les millions d'instructions par seconde. Voir http ://www.mips.com/

TABLEAU 1.2 Les différents simulateurs offerts par SimpleScalar

Nom du simulateur	Description	Vitesse
sim-safe	Simulateur fonctionnel simple	6 MIPS
sim-fast	Simulateur fonctionnel optimisé pour la vitesse	7 MIPS
sim-profile	Simulateur avec profileur dynamique	4 MIPS
sim-bpred	Simulateur de prédiction de branchement	5 MIPS
sim-cache	Simulateur de hiérarchie de la mémoire cache	4 MIPS
sim-fuzz	Générateur d'instructions aléatoires et testeur	2 MIPS
sim-outorder	Simulation détaillée de l'architecture et des modèles de délais	0.3 MIPS

d'affirmer que l'exploration avec le Xtensa se limite à des raffinements de son ISA de base qui peut être décrit, en deux mots, comme étant un DLX étendu. Avec les nouvelles versions de SimpleScalar, de nouveaux modèles de processeurs sont cependant supportés. Plus précisément, des modèles de processeurs Alpha, ARM, PowerPC et x86 sont disponibles [6]. Afin de supporter un nouveau processeur, les couches supérieures de la figure 1.4, qui montre l'architecture du simulateur, doivent être modifiées.

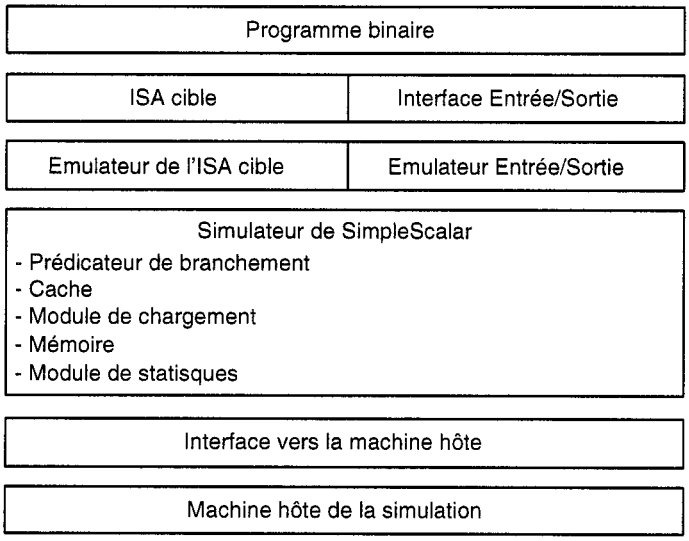


FIGURE 1.4 Architecture du simulateur SimpleScalar

Les deux couches de base de la figure 1.4 permettent de rouler SimpleScalar sur différents types de machines. Le noyau du simulateur contient, quant à lui, plusieurs modules qui sont génériques aux ISS. Par exemple, on y retrouve des modèles de cache et de mémoire, un module pour charger le code binaire en mémoire et un module de statistiques afin de faire du profilage. Ce noyau est aussi responsable d'analyser, cycle par cycle, chacune des lignes de code du programme et d'en simuler le bon comportement. Pour ce faire, il utilise les couches supérieures de SimpleScalar afin d'interpréter le jeu d'instructions et de simuler les entrées et sorties du processeur. Pour simuler un nouveau processeur, il est donc nécessaire de réécrire le code pour ces deux couches supérieures. Puisque le code source de SimpleScalar est disponible, il est bien entendu possible de l'utiliser comme base pour réaliser des simulations qui n'étaient pas initialement adaptées à l'outil et ainsi explorer différents jeux d'instructions. Un manuel d'instructions pour expliquer les étapes à suivre est d'ailleurs disponible [5].

Grâce à cette flexibilité, SimpleScalar peut être utilisé pour valider des nouvelles instructions. Dans l'article [90], les auteurs ont porté la pile TCP/IP du système d'exploitation FreeBSD pour qu'il roule sur SimpleScalar avec un ISA inspiré du MIPS (PISA). Ils ont pu étudier plusieurs configurations de mémoire cache afin de trouver la plus optimale. De plus, ils ont profilé l'application afin de déterminer les paires d'instructions les plus utilisées (par exemple une addition suivie d'un branchement) et de créer une nouvelle instruction qui étend l'ISA dans SimpleScalar. Des gains de l'ordre de 23% on été observés. L'exemple décrit ci-dessus illustre bien que SimpleScalar peut être utilisé pour explorer différents jeux d'instructions, et avec son infrastructure flexible de simulation, il offre un niveau de flexibilité semblable à LISATek. Cependant, contrairement au Xtensa et à LISATek, aucune voie vers une réalisation physique du processeur n'est disponible. De plus, si l'on désire modifier un jeu d'instructions d'un processeur qui n'est pas supporté par SimpleScalar, il est nécessaire de commencer par ajouter ce modèle d'ISA dans le code de SimpleScalar.

Cette tâche est plus simple que réécrire son propre ISS du début, puisque SimpleScalar fournit déjà une bonne infrastructure, mais elle reste non triviale. Un groupe de recherche de l'université de Toronto [52] travaille d'ailleurs sur un outil qui automatise l'ajout d'un nouvel ISA dans SimpleScalar. Le nouveau processeur est décrit à l'aide d'un langage nommé Babel. Ce langage, qui n'est pas sans analogies avec LISA, permet de décrire l'ISA ainsi que l'interface binaire (ABI) du nouveau processeur. À partir d'une description Babel, le processeur peut être automatiquement intégré dans SimpleScalar.

Bien qu'il ait des limitations, SimpleScalar reste un outil intéressant pour l'exploration architecturale et il est développé activement aussi bien par la compagnie SimpleScalar LCC que par la communauté scientifique. Plusieurs extensions sont disponibles comme, par exemple, Wattch [12] qui permet d'obtenir rapidement un estimé de la puissance consommée par un processeur. Il s'agit là d'un avantage de SimpleScalar vis-à-vis LISATek. Il est prévu que dans des versions futures, les simulations multiprocesseurs et multiprocessus (HMT) seront supportées. Notons finalement que SimpleScalar vise principalement à modéliser des architectures existantes et n'offre aucun support pour la réalisation matériel d'un processeur.

1.1.4 Autres types de processeurs configurables

Le domaine des processeurs configurables est présentement en pleine effervescence et plusieurs autres compagnies offrent des produits dans ce domaine. ARC, Critical Blue, Stretch et Improv Systems sont tous des compagnies qui offrent une ou des solutions configurables. L'annexe I offre plus de détails sur les produits de ces différentes compagnies.

1.2 Différentes techniques d'exploitation du parallélisme

Utiliser des processeurs configurables avec des jeux d'instructions spécialisés constitue une approche nouvelle et prometteuse pour augmenter les performances d'un système sur puce. Il s'agit d'une option particulièrement intéressante vu la possibilité de rapidement créer un tel processeur en utilisant les technologies présentées à la section 1.1. Cependant, les processeurs configurables ne permettent pas nécessairement de résoudre un des principaux problèmes dans les systèmes-sur-puce : la latence des communications. En effet, un système-sur-puce d'une bonne complexité va typiquement comprendre plus d'un processeur, des mémoires, des coprocesseurs et des modules d'entrée et sortie. Toutes ces composantes sont habituellement reliées ensemble à l'aide d'un canal de communication qui peut offrir des caractéristiques typiques d'un réseau de communication (*split transaction*, chemins multiples, gestion et arbitrage du trafic), c'est pourquoi nous les appelons souvent réseau sur puce (*Network on chip* ; NoC) au lieu de bus partagé. Bien que ces réseaux soient performants et que les mémoires embarquées dans la puce soient rapides, les processeurs se retrouvent souvent à attendre plusieurs cycles pour faire des requêtes sur le canal de communication. Ces délaient s'avèrent un problème de taille lorsque nous voulons traiter des informations à un haut débit, comme c'est le cas dans un processeur réseau.

Il est donc très important d'utiliser des techniques pour soit diminuer ou masquer cette latence, tout en sachant qu'il n'est pas possible de réduire à zéro les délais dans le canal. Un bon nombre de techniques permettent de s'approcher cet objectif, les mémoires cache en sont une des plus utilisées pour diminuer la latence. Une autre technique efficace est de masquer la latence en utilisant plusieurs fils d'exécution concurrents sur le processeur. Lorsqu'un processeur bloque, il est possible de changer de processus pendant que le premier attend. Utiliser une telle technique permet non seulement de masquer la latence du canal, mais aussi de traiter les informations en parallèle, ce qui a toujours été une des techniques les plus efficaces pour augmenter

les performances d'un circuit.

Il existe plusieurs techniques pour exploiter le parallélisme d'une application [53, 89] et s'assurer par le fait même que le taux d'utilisation des processeurs du système sur puce soit le plus élevé possible et ne soit pas affecté par la latence des communications. Les plus communes sont présentées dans le reste de cette section.

1.2.1 Processeur RISC classique

Un processeur RISC classique pour systèmes embarqués comme le ARM, le PowerPC 405, ou le Xtensa (section 1.1.1) possède un petit pipeline capable de traiter une nouvelle instruction par cycle dans les conditions idéales : c'est-à-dire lorsqu'aucune dépendance de données n'existe et qu'aucun accès mémoire ne risque de bloquer le pipeline. Dès qu'un processeur de ce type effectue plusieurs accès sur le canal de communication, son pipeline va être bloqué et des bulles vont être insérées, ce qui fait chuter le pourcentage d'utilisation bien en deçà de l'idéal. Ce type de processeur ne possède aucun mécanisme pour rapidement changer de fil d'exécution lorsqu'il est bloqué. De plus, il n'est pas capable d'extraire du parallélisme à l'intérieur même du processus, ni de lancer plus d'une instruction par cycle. La simplicité de ce processeur peut s'avérer un avantage dans certaines applications, vu sa taille et sa consommation de puissance réduite.

1.2.2 Processeur superscalaire

Un processeur superscalaire est une première solution qui permet d'exploiter le parallélisme d'une application. C'est un processeur qui possède plusieurs unités d'exécution parallèles, ce qui lui permet de lancer plus d'une instruction à chaque cycle [33, 79]. Afin de garder ses multiples pipelines le plus occupé possible, le processeur superscalaire est doté de logique qui extrait le parallélisme des instructions d'un

même processus. Ce processeur est capable de déterminer où sont les dépendances entre les instructions et, au besoin, d'exécuter les instructions dans le désordre (si les dépendances le permettent) afin de garder ses pipelines le plus occupé possible. À la sortie des pipelines, les instructions sont réordonnancées afin que les registres logiques soient mis à jour correctement. Ce type de processeur possède aussi plus de registres physiques que de registres logiques et lorsqu'un groupe d'instructions à exécuter dans le désordre est analysé, les registres logiques sont renommés sur des registres physiques. Cette opération permet ainsi d'éliminer des fausses dépendances causées par le manque de registres. Ce type de parallélisme au niveau des instructions est nommé ILP (*instruction level parallelism*) ou encore parallélisme horizontal et il est capable de masquer la latence des opérations plus lentes dans la mesure ou le processeur reste capable de remplir ses pipelines. Bien qu'il s'agisse d'une architecture assez complexe qui peut demander beaucoup de surface et de puissance, elle offre des performances intéressantes et la plupart des microprocesseurs modernes tels ceux créés par Intel ou AMD appartiennent à cette catégorie.

1.2.3 Processeur avec traitement multiprocessus matériel

Ce type de processeur, couramment appelé *Hardware MultiThreaded* (HMT), offre la possibilité de traiter plusieurs processus concurrents sans passer par un système d'exploitation. Plus précisément, un processeur HMT contient les registres nécessaires (plusieurs compteurs de programme, pointeurs vers la pile, registres d'états et de données) pour sauvegarder l'état de chacun des fils d'exécution concurrents. À chaque cycle, le processeur peut changer de contexte et passer d'un fil d'exécution à un autre sans délai puisqu'aucune manipulation sur les registres n'est nécessaire. Cela contraste avec un changement de contexte effectué par du logiciel tel un système d'exploitation roulant sur un processeur simple, puisque ce dernier demande un grand nombre de cycles pour sauvegarder l'état de l'ancien processus et charger le nouveau. Un changement de contexte logiciel peut entraîner une latence encore plus grande

que le délai de communication qui devrait être masqué, d'où la nécessité d'utiliser un processeur avec support pour le HMT. Le parallélisme exploité par un processeur HMT est appelé parallélisme vertical ou encore TLP (*thread level parallelism*). Le concept du HMT peut être appliqué sur différents types de processeurs et il peut être appliqué de différentes manières. Il en résulte une gamme de nuances relativement variées.

Parallélisme à grain fin. Le processeur change de processus à chaque cycle dans un ordre prédéterminé et boucle sur cette liste (ordonnancement de type tourniquet). En plus d'être simple, cette approche permet de minimiser les dépendances dans le pipeline puisque la deuxième instruction d'un même processus est insérée dans le pipeline seulement N cycles après la première, ou N est le nombre de processus concurrents supportés par le processeur. Puisque le processeur change de contexte à chaque cycle, il est bien adapté pour masquer les latences de moyenne longueur (moins de N cycles) et aussi diminuer l'impact des longues latences. Bien entendu ce type de processeur n'est pas apte à exécuter un seul programme qui s'exécute à l'intérieur d'un seul processus puisque, dans ce cas, le processeur sera inactif la plupart du temps ($(N-1)/N$ cycles) et le débit total sera très bas. Ce type de processeur dispose d'une architecture très simple, ce qui explique pourquoi le HMT était utilisé dans les premiers superordinateurs des années 1980. Le Tera, un des premiers processeurs HMT, se trouve dans cette catégorie avec son support pour 128 processus concurrents et un pipeline très profond [3]. Le processeur HEP [78] et le processeur MASA [28] sont aussi des exemples anciens de cette technologie.

Parallélisme à gros grain. Avec ce type de HTM, le changement de contexte s'effectue lors d'un événement prédéterminé qui ajouterait des délais dans le pipeline, par exemple un échec d'accès en mémoire cache, une mauvaise prédiction de branchement ou encore un accès sur le bus externe. Bien qu'il puisse se faire en un seul cycle si l'architecture le supporte, le changement de contexte dans ces processeurs est typiquement plus long que dans les HMT à grain fin, car souvent un seul processus

peut être dans le pipeline à la fois. Ces processeurs sont donc mieux adaptés pour masquer les longues latences que les petites. De plus, ce mode de changement de contexte offre des performances raisonnables lorsqu'un seul processeur est en cours d'exécution. Le processeur APRIL qui peut effectuer un changement de contexte en 4 à 10 cycles est un des premiers exemples de cette technologie [2]. Un autre exemple plus récent est le processeur HMT à gros grain basé sur un PowerPC et utilisé dans le AS/400 de IBM [80]. Ce processeur supporte 2 processus concurrents et effectue des changements de contexte en 3 cycles. Sun a aussi créé une architecture multiprocesseur, le MAJC (2 processeurs sur une puce), qui supporte le HMT à gros grain sur chacun des processeurs [85].

Type de pipeline. Un processeur avec support HMT peut aussi bien avoir un pipeline simple, qu'un pipeline superscalaire. Avec un pipeline simple, le processeur peut exploiter le parallélisme vertical et ainsi masquer la latence des communications. Dans le cas du pipeline superscalaire, le processeur peut lancer plus d'une instruction du processus à chaque cycle. Ce type de processeur est donc capable d'exploiter le parallélisme horizontal aussi bien que le vertical. Ces deux architectures de pipeline peuvent aussi bien être utilisées avec un parallélisme à grain fin qu'à gros grain.

1.2.4 Multiprocesseurs sur puce

L'idée la plus simple pour augmenter le nombre de processus qui peuvent être exécuté en parallèle est de mettre plusieurs petits processeurs sur la même puce et de distribuer (statiquement dans le cas simple ou dynamiquement sinon) les processus sur chacun des noyaux. Habituellement, avec une approche CMP (*on-Chip MultiProcessor*), le nombre de processus qui roule sur chaque processeur est diminué et ils sont plutôt distribués sur plusieurs processeurs. Il en résulte un circuit simple à concevoir puisqu'il est basé sur des processeurs existants et aussi un processeur avec des performances comparables au HMT tel que démontré dans [30]. Cependant, un processeur

CMP demande plus de surface car les unités fonctionnelles ne sont pas partagées comme c'est le cas avec le HMT. Un autre désavantage potentiel du CMP est que si un seul processus est exécuté, alors les ressources sont sous-utilisées, puisqu'un seul des noyaux sera actif. Bien entendu, utiliser plusieurs processeurs sur une puce n'exclut pas les autres techniques HMT décrites ci-dessus, ni l'utilisation de processeurs superscalaires. Ces optimisations sont plutôt orthogonales, le MAJC, qui exploite les deux techniques, illustre bien ce fait. Dans un même ordre d'idées, [56] montre qu'un circuit avec 4 petits processeurs pouvant exécuter 2 instructions par cycle, chacun offre des performances meilleures ou comparables à un processeur superscalaire pouvant traiter 6 instructions par cycle. Le processeur Power4 de IBM contient deux processeurs superscalaires par circuit ainsi que des interfaces spécialisées pour connecter plusieurs Power4 ensemble [83]. Le processeur UltraSparcIV de Sun [81] offre de son côté des caractéristiques semblables au Power4.

1.2.5 Processeur superscalaire avec multiprocessus

Les gains offerts par un processeur superscalaire régulier peuvent être améliorés en augmentant le nombre d'instructions qu'il est possible de lancer à chaque cycle et en augmentant la taille de la mémoire tampon qui contient les instructions à réordonnancer. Cependant, passé un certain stade, les gains commencent à être moins intéressants et la quantité de registres nécessaires ainsi que la complexité du circuit deviennent très grandes. C'est là un des principaux arguments du CMP face au superscalaire [56]. Par exemple, il n'est pas très utile de pouvoir lancer plus de 8 instructions en même temps puisqu'un processeur avec 8 unités fonctionnelles réussit en moyenne à en occuper seulement 40% pour un CPI de 3.2 [86]. D'un autre côté, les tampons d'instructions à réordonnancer sont rendus assez larges (126 dans le cas d'un Pentium 4) que les augmenter n'apporte plus suffisamment d'avantages. Il devient donc difficile d'extraire plus de parallélisme au niveau des instructions (ILP) avec un processeur superscalaire standard.

L'idée du processeur superscalaire avec multiprocessus (*Simultaneous Multithreading* ; SMT) est de combiner les forces du processeur superscalaire avec un support matériel pour traiter plusieurs fils d'exécutions concurrents tout en restant très flexible, contrairement au HMT. C'est-à-dire qu'à chaque cycle, le processeur peut aller chercher ses instructions dans n'importe quel des processus concurrents qu'il supporte. Lorsque le tampon d'instructions à ordonnancer doit être rempli, le processeur peut par exemple choisir des instructions en provenance du processus qui en a le moins dans le tampon, ce qui permet aux processus rapides d'avancer et garanti qu'il n'y aura pas de famine. Avec ce mode de fonctionnement, le processeur SMT peut beaucoup plus facilement remplir ses unités fonctionnelles à chaque cycle et les ressources du processeur sont partagées dynamiquement à travers tous les processus. Tout comme le processeur HMT, le SMT possède un compteur de programme et autres registres d'états pour chacun des processus. Il est aussi doté d'un plus grand nombre de registres qu'un superscalaire standard ainsi que d'une unité de prédiction de branchement avec des identificateurs de processus. Il a été montré dans [86, 16] que dans un grand nombre d'applications cette approche est celle qui offre les plus grandes performances par rapport au superscalaire, au HMT et au CMP à cause de sa grande flexibilité. D'un autre côté, [30, 29] arrivent à la conclusion inverse et favorise le CMP à cause de son design plus simple, surtout au niveau des mémoires caches.

En effet, un processeur SMT offre des performances intéressantes mais apporte aussi des défis lors de la conception du matériel. Par exemple, étant donné le grand nombre d'instructions lancées à partir de plusieurs processus à chaque cycle, ce processeur demande beaucoup à la mémoire cache, tant au niveau des tailles que de la bande passante. Il en va de même pour les tables de prédiction de branchement. Bien que les notions de SMT soient discutées et analysées depuis le milieu des années 90, jusqu'à tout récemment, peu de processeurs commerciaux utilisaient cette technologie ; principalement parce qu'il s'agit de circuits compliqués à réaliser et qui demandent

beaucoup de ressources. Le Pentium 4 avec Hyper Threading de Intel [48] qui supporte 2 processus concurrents sur l'architecture superscalaire du Pentium 4 classique est un des premiers processeurs commerciaux avec SMT. D'autres projets commerciaux s'intéressent au SMT, par exemple IBM avec son Power5 [39] et la compagnie Sun avec le processeur du nom de code Niagara viennent de sortir ou ont annoncé la mise en marché de processeurs SMT dans un avenir rapproché.

1.2.6 Résumé des différentes techniques

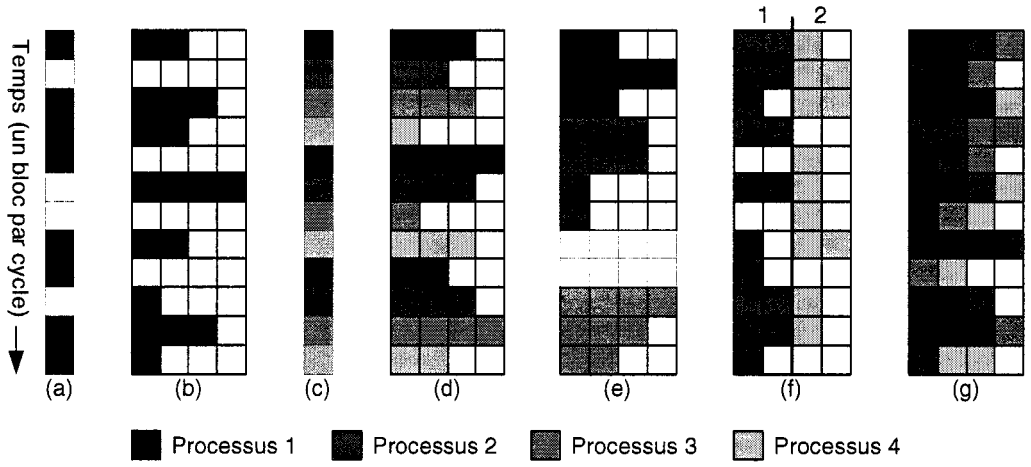


FIGURE 1.5 Résumé des différentes architectures de processeurs avec traitement multiprocessus

La figure 1.5 présente un résumé des différentes architectures de processeurs présentées et les formes de parallélisme qu'ils utilisent [38]. Chaque case représente une unité fonctionnelle sur laquelle il est possible de lancer une instruction et chaque ligne représente un cycle d'horloge. Une case blanche indique qu'il n'y a pas eu d'instruction lancée à ce cycle (une bulle). Ces graphiques illustrent bien le parallélisme horizontal (ILP) et le parallélisme vertical (TLP) exploités par chacun des processeurs. Plus précisément les sept figures illustrent :

a) Un processeur RISC simple qui ne traite qu'un processus à la fois. Les dépendances de données ainsi que les accès mémoires empêchent le processeur

de lancer une nouvelle instruction à chaque cycle.

b) Un processeur superscalaire avec 4 unités fonctionnelles. Ce processeur extrait du parallelisme horizontal, mais n'est pas très bien adapté pour masquer la latence des accès sur les canaux de communication.

c) Un processeur HMT à grain fin qui passe d'un processus à l'autre à chaque cycle en mode tourniquet. Ce processeur est optimisé pour traiter un nombre fixe de processus concurrents.

d) Un processeur HMT à grain fin avec pipeline superscalaire.

e) Un processeur HMT à gros grain avec pipeline superscalaire. Les changements de contextes se font sur des événements prédéterminés et peuvent demander quelques cycles dépendamment de l'architecture.

f) Un processeur CMP avec deux noyaux. Dans cet exemple chaque noyau à des capacités superscalaire limitées (2 instructions par cycle).

g) Un processeur SMT capable d'aller chercher ses instructions dans n'importe quel processus à chaque cycle.

L'approche SMT offre d'excellentes performances et plusieurs fabricants de microprocesseurs se sont lancés dans cette direction en combinant le SMT avec du support pour multiprocesseur ou pas. Cela semble la voie du futur pour les processeurs utilisés dans les ordinateurs de bureau et les serveurs. Cependant, dans le cas des systèmes sur puce qui sont souvent appelés à être intégrés dans un système embarqué de taille réduite et fonctionnant sur une pile (tel un téléphone cellulaire), ces processeurs sont trop coûteux en terme de surface et de consommation de puissance pour être une solution intéressante en ce moment. De plus, les articles [20, 86, 16] tendent à prouver que le SMT est plus performant pour des applications typiquement utilisées sur un ordinateur de bureau. Ce n'est pas nécessairement le cas pour des applications embarquées précises comme pour les processeurs réseaux. Certaines recherches montrent aussi que pour exploiter le SMT au maximum, il est nécessaire de programmer le code en conséquence [49]. De plus, dans un SoC, le fait d'utiliser des techniques

CMP est assez orthogonal à la technologie utilisée dans chacun des processeurs du système. C'est pourquoi ce mémoire se concentrera surtout sur le HMT sans support superscalaire. Une approche multiprocesseur pourra être intégré dans une étape ultérieure. Notons finalement qu'utiliser un processeur supportant plusieurs processus concurrents n'est pas la seule approche possible pour diminuer la latence des communications, l'annexe II décrit brièvement certaines de ces autres approches.

1.3 Les processeurs réseaux

Toutes les techniques présentées à la section 1.2 pour exploiter le parallélismes sont surtout intéressantes lorsque les processeurs de la plate-forme sont appelés à travailler sur une application qui exploite un grand nombre de processus concurrents. De même, la création d'instructions spécialisées et la configuration des processeurs doivent se faire dans le cadre d'une classe d'application avec ses caractéristiques propres. Le traitement de paquets réseaux ainsi que les applications multimédias sont deux domaines où les techniques décrites dans ce chapitre peuvent être appliquées avec beaucoup de succès [16]. Dans le cadre de ce travail, les applications réseaux et les processeurs réseaux ont été étudiés en plus grande profondeur et ont servi d'exemples pour valider notre méthodologie. La présente section décrit donc brièvement ce qu'est un processeur réseau ainsi que ses caractéristiques architecturales.

1.3.1 Définition

Une définition très générale d'un processeur réseau serait tout circuit capable de traiter efficacement des paquets d'un réseau de communication. En pratique, si l'on regarde les circuits existants vendus sous le nom de processeur réseau, ou NPU (*Network Processing Unit*), il s'agit d'un système sur puce qui comprend de la logique

programmable (des microprocesseurs) afin de répondre au besoin de flexibilité, et des unités de calculs spécialisées afin de pouvoir traiter un grand nombre de paquets par seconde. Les défis à relever lors de la création d'un NPU sont :

- la possibilité de transférer les paquets dans le NPU, de les traiter et de les sortir sur le réseau à un très haut débit (plusieurs Gbit/s) ;
- la grande quantité de bande passante et de mémoire nécessaire à l'intérieur du circuit ;
- la capacité de manipuler des groupes de bits de tailles variables et pas nécessairement alignés sur la mémoire ;
- la capacité d'éclipser la latence des différentes opérations à réaliser sur un paquet.

Un NPU est donc un circuit monopuce spécialisé pour effectuer du traitement sur des paquets d'un réseau de communication. Un routeur, quant à lui, peut être composé d'un ou de plusieurs processeurs réseaux (sur différentes cartes de ligne par exemple) ainsi que d'autres modules essentiels comme des interfaces physiques. Une description plus détaillée de l'architecture des processeurs réseaux, des applications qu'ils exécutent, des protocoles utilisés ainsi que du modèle OSI (*Open System Interconnection*) peut être trouvée dans [68]. La présente section ne vise pas à entrer dans ces détails et se limite plutôt à donner un survol des différentes caractéristiques des processeurs réseaux existants.

1.3.2 Caractéristiques

Il existe un grand nombre de modèles de processeurs réseaux commerciaux. Le document [75] offre un excellent tour d'horizon des différents produits existants et de leurs architectures et caractéristiques respectives. Les NPU utilisent en grande majorité certaines des techniques présentées précédemment (HMT, CMP) afin d'être capable de traiter les paquets à haut débit. Plus précisément, un processeur réseau comprend typiquement plusieurs des caractéristiques suivantes :

- une architecture multiprocesseur ;

- des coprocesseurs matériels ;
- des processeurs capables de traitement multiprocessus ;
- un jeu d'instructions spécialisé ;
- des réseaux d'interconnexions sur puce rapide (NoC) ;
- une structure de mémoire avancée.

Nous décrivons chacune de ces techniques plus en détail ci-dessous, tout en donnant quelques exemples de NPU commerciaux les exploitants.

1.3.2.1 Architecture multiprocesseur

Un NPU peut facilement tirer avantage d'une architecture multiprocesseur, puisque le traitement d'un paquet est très souvent indépendant des autres paquets dans le routeur. Grâce à ce parallélisme inhérent à l'application, il est naturel de séparer le flot de données sur différents processeurs et ainsi augmenter le débit. Par exemple, le Motorola C-5e[4] utilise un processeur général pour le contrôle et 16 petits processeurs spécialisés pour travailler sur les paquets. Chaque processeur spécialisé est en fait un RISC avec deux modules pour le transfert de données (un pour la réception et un pour l'émission de paquets). Le IPX2800 de Intel[5] possède lui aussi 16 petits processeurs RISC pour le traitement des paquets et un processeur embarqué pour la logique de contrôle. Le NP4GS4 de IBM[6] offre, encore une fois, une architecture similaire avec ses 16 processeurs de protocole et son PowerPC 405 pour le contrôle. D'autres architectures sont cependant nettement différentes. Alors que dans les trois exemples précédents, chaque processeur était symétrique et pouvait effectuer le même traitement sur différent paquets, le NP-1 de EZChip[7] opte plutôt pour un grand nombre de processeurs (jusqu'à 64), qui effectuent chacun une tâche précise sur

[4]http ://www.motorola.com
[5]http ://www.intel.com
[6]http ://www.ibm.com
[7]http ://www.ezchip.com

le paquet tel un pipeline. Le PayloadPlus de Agere[8] utilise aussi la technique du pipeline, mais avec seulement 3 processeurs plus complexes.

1.3.2.2 Coprocesseurs

Bon nombre de NPU possèdent des processeurs avec un jeu d'instructions adapté pour le traitement de paquets. Cependant, cela ne s'avère pas suffisant pour pouvoir travailler avec les hauts débits demandés. Les NPU utilisent donc du matériel spécialisé pour certaines tâches qui ne peuvent être implémentée efficacement en logiciel. Des coprocesseurs sont souvent utilisés pour remplir les fonctions suivantes :

- la recherche dans une table de routage ;
- la gestion de queues de paquets ;
- calculer certains champs du paquet tel le checksum (un code de détection d'erreur) ;
- le déplacement de paquets à l'intérieur du circuit (par exemple pour copier un paquet en mémoire) ;
- l'authentification et le chiffrement des données.

La vaste majorité des NPU utilisent des coprocesseurs, par exemple, le 5-Ce en contient 5 (3 pour les communication, un pour la table de routage et un pour les queues de paquet). De son côté, le IPX2800 possède aussi plusieurs modules matériels dont un pour le support de IPSec (protocole d'authentification et de chiffrement).

1.3.2.3 Traitement multiprocessus

Le fait que plusieurs NPU disposent des ressources matérielles pour traiter plusieurs processus concurrents, tel que décrit à la section 1.2, est un bon indicateur des avantages non négligeables de cette approche. Sur le NP4GS3, chacun des 16 processeurs supporte 2 processus et un changement de contexte s'effectue lors d'un accès

[8]http ://www.agere.com

bloquant. Le IPX2800 supporte aussi plusieurs processus sur chacun de ces microprocesseurs. Certains NPU, dont le PXF/Toaster 2 de Cisco, utilisent une architecture VLIW pour prendre avantage du parallélisme au niveau des instructions (ILP). Le CNP810SP de ClearWater Networks (qui n'a jamais été fabriqué puisque la compagnie à fait faillite) utilise une approche différente et est construit autour d'un seul processeur SMT capable de lancer 10 instructions par cycle et supportant 8 processus concurrents. Des recherches académiques ont aussi montrés que le SMT pouvait être intéressant dans les NPU [87]. Le MPS5000 de Brecis[9] utilise 2 processeurs superscalaires capables de lancer 4 instructions par cycle. Il reste que la majorité des NPU existant ont optés pour un grand nombre de processeurs simples plutôt qu'un petit nombre de processeurs superscalaires puisque le TLP offre des gains plus intéressants que le ILP pour les processeurs réseaux.

1.3.2.4 Jeu d'instructions spécialisé

Tel que mentionné précédemment, plusieurs NPU possèdent des processeurs avec un jeu d'instructions adapté au traitement de paquets. Par exemple le IPX2800, possède une instruction permettant de trouver le premier bit à 1 dans un registre en un cycle. Très souvent les instructions spécialisées dont les fabricants parlent sont une interface afin que le processeur envoie sa requête au matériel spécialisé que l'on trouve dans les coprocesseurs [76]. Il ne s'agit donc pas d'instructions spécialisées plus complexes comme il est possible d'en créer avec le Xtensa.

1.3.2.5 Interconnexions

Pour les NPU où les processeurs fonctionnent comme un pipeline, les interconnexions sont surtout point à point. Mais pour l'architecture plus commune où les processeurs

[9]http ://www.brecis.com

sont symétriques, le réseau d'interconnexions doit supporter un haut débit. Afin d'atteindre la bande passante voulue, les fabriquants développent leur propre architecture de bus ou encore utilisent plusieurs bus (le 5-Ce en possède 5).

1.3.2.6 Structure mémoire

Les NPU existants utilisent généralement une architecture de mémoire relativement complexe avec plus d'une mémoire sur la même puce. Un patron de type NUMA tel que décrit à l'annexe II est parfois utilisé.

CHAPITRE 2

MÉTHODOLOGIE ET OUTILS

Afin de réaliser notre NPU utilisant un processeur avec des instructions spécialisées, il est nécessaire de se doter d'outils de design et aussi d'une méthodologie de conception. Ce chapitre débute donc par une brève revue du design au niveau système et des différents outils de design à haut niveau d'abstraction, tant commerciaux qu'académiques. Par la suite, les outils employés pour ce travail, c'est-à-dire StepNP, SystemC et Click, seront présentés.

Un autre élément nécessaire pour modéliser, dans StepNP, un NPU qui utilise un processeur configurable est l'intégration de l'ISS de Xtensa dans StepNP. L'avant dernière partie de ce chapitre présentera donc l'intégration d'un ISS dans une simulation SystemC et plus particulièrement le Xtensa dans StepNP. Finalement, la méthodologie de codesign employée pour cette recherche sera abordée.

2.1 Design au niveau système

Tel qu'expliqué dans l'introduction, il est devenu nécessaire de disposer d'un modèle haut niveau simulable d'un SoC en développement le plus tôt possible. Cela permet d'obtenir rapidement des estimés de performance, de fixer l'architecture de base de la plate-forme et de valider le logiciel qui s'exécutera sur la plate-forme longtemps avant d'avoir un modèle plus détaillé. De plus, effectuer des modifications sur un modèle simulable à haut niveau d'abstraction est beaucoup moins coûteux, en terme de temps et d'argent, que d'effectuer des modifications lorsque le SoC est prêt à être fabriqué. Un autre avantage non négligeable d'un modèle haut niveau est qu'il offre des vitesses de simulation qui sont plusieurs ordres de grandeurs plus rapides qu'un

modèle bas niveau (comme du code RTL). La boucle de rétroaction qui indique au développeur si l'architecture simulée est prometteuse ou non est donc plus rapide, ce qui permet d'explorer un plus grand nombre d'architectures.

Plusieurs langages permettant de faire du design au niveau système ont vu le jour récemment. La bibliothèque SystemC qui étend les fonctionnalités du C++ figure parmi les plus populaires présentement. SystemC est le langage utilisé pour cette recherche et sera décrit plus en détail à la section 2.2.1. D'autres langages tels SpecC [24] et SystemVerilog offrent aussi des possibilités de design au niveau système qui sont comparables à celles de SystemC. Bien que très utile pour modéliser un design au niveau système, SystemC n'est pas très bien adapté pour la modélisation du logiciel. De plus, SystemC est uniquement un langage de modélisation et ne fournit pas non plus, à proprement parler, une méthodologie de conception, malgré qu'une certaine méthodologie pour une modélisation au niveau transactionnelle soit décrite dans [27].

Il est donc fort utile de se doter d'une méthodologie et d'outils lorsque l'on désire réaliser un modèle de SoC en SystemC. Plusieurs outils tentent déjà d'automatiser, ou du moins faciliter, le design des SoC. Pour la réalisation de cette recherche, nous avons utilisé un outil de conception à haut niveau nommé StepNP (décrit à la section 2.2). Il existe cependant un grand nombre d'outils de développement de plateforme et d'exploration architecturale, tant commerciaux qu'académiques. Certains des plus connus sont brièvement nommés ci-dessous ; une liste plus complète peut être retrouvée dans [26, 22].

Avec la popularité grandissante des langages de design au niveau système, plusieurs compagnies offrent aujourd'hui des outils pour faciliter ce type de conception. Coware, Prosilog, Summit Design et Synopsys offrent tous des outils qui peuvent faciliter le design en SystemC en automatisant certaines tâches. Ces outils, ainsi que d'autres offerts par Cadence et Mentor Graphics, permettent aussi de faire des co-

simulations entre différents langages et de l'exploration architecturale. Du côté des outils académiques le choix est encore plus grand. Ptolemy, MESCAL, Metropolis et Roses sont parmi les plus connus. Le groupe de recherche en microélectronique de l'École Polytechnique de Montréal développe aussi son propre outil d'exploration nommé SPACE. Une description plus détaillée de tous ces outils est disponible à l'annexe III. Le domaine du codesign logiciel/matériel et du développement de SoC à haut niveau d'abstraction est donc un domaine de recherche encore très actif.

2.2 La plate-forme de développement StepNP

Pour ce travail, la plate-forme d'exploration architecturale utilisé se nomme StepNP (*System-level Exploration Platform for Network Processing*). Plusieurs raisons ont motivé le choix de StepNP. Tout d'abord, il s'agit d'un ensemble d'outils complet pour le développement d'une architecture à haut niveau. En second lieu, StepNP est doté d'une bibliothèque comprenant plusieurs blocs (tels des processeurs, des mémoires et des interconnexions) qui peuvent facilement être connectés ensemble. De plus, les architectures créées à l'aide de ces blocs peuvent être facilement modifiées en ajoutant ou en adaptant des composantes. Finalement, le code source de StepNP est fourni gratuitement par ST Microelectronics pour la recherche universitaire, ce qui offre de belles possibilités de collaborations.

StepNP est donc un outil basé sur SystemC qui, comme son nom l'indique, se penche sur l'exploration architecturale des processeurs réseau [60]. Notons que plus récemment les types d'architectures développées avec StepNP se sont diversifiés pour aussi inclure les applications multimédia telles la compression et l'encodage de vidéo. En quelques mots, StepNP peut être décrit comme un environnement de développement pour les SoC multiprocesseurs qui fournit une bibliothèque de composants standards prêts à être intégrés ensemble, ainsi que des interfaces bien définies pour créer et connecter des nouveaux composants. StepNP fournit aussi un environ-

nement de simulation, un environnement de développement pour le logiciel embarqué et des outils d'analyse et de profilage.

Tel qu'illustré à la figure 2.1, StepNP se divise en trois grandes parties :

a) un modèle d'architecture mono ou multiprocesseur décrit en SystemC ;

b) un environnement de développement de logiciel basé sur Click ;

c) des outils d'introspection, d'analyse et de profilage du modèle SystemC.

Chacune de ces trois parties, ainsi que la bibliothèque SystemC, seront décrites dans les prochaines sous-sections. Pour les lecteurs intéressés à utiliser StepNP, un manuel de référence plus complet, incluant un tutorial et un guide d'utilisation, est présentement rédigé par la Société Canadienne de microélectronique[1] (CMC/SCM) et devrait être disponible sous peu.

2.2.1 Utilisation de SystemC dans StepNP

À la base de StepNP, nous retrouvons la bibliothèque de modélisation de matériel SystemC version 2.0.1. Il est donc nécessaire de posséder une bonne connaissance de cette bibliothèque pour utiliser StepNP. Une description détaillée de SystemC et de son fonctionnement dépasse le cadre de ce mémoire, le lecteur intéressé pourra consulter plusieurs ouvrages sur le sujet [27, 10, 57, 58, 59, 82]. De plus, l'annexe IV donne un bref aperçu des niveaux d'abstractions offerts par SystemC dont le niveau TLM (*Transaction Level Modeling*) qui est très utilisé dans StepNP.

2.2.2 Modèle d'architecture et bibliothèque de composants

À la base de StepNP, nous retrouvons une architecture simulable décrite en SystemC qui utilise un mécanisme de communication TLM. Cette architecture est créée par

[1] http ://www.cmc.ca

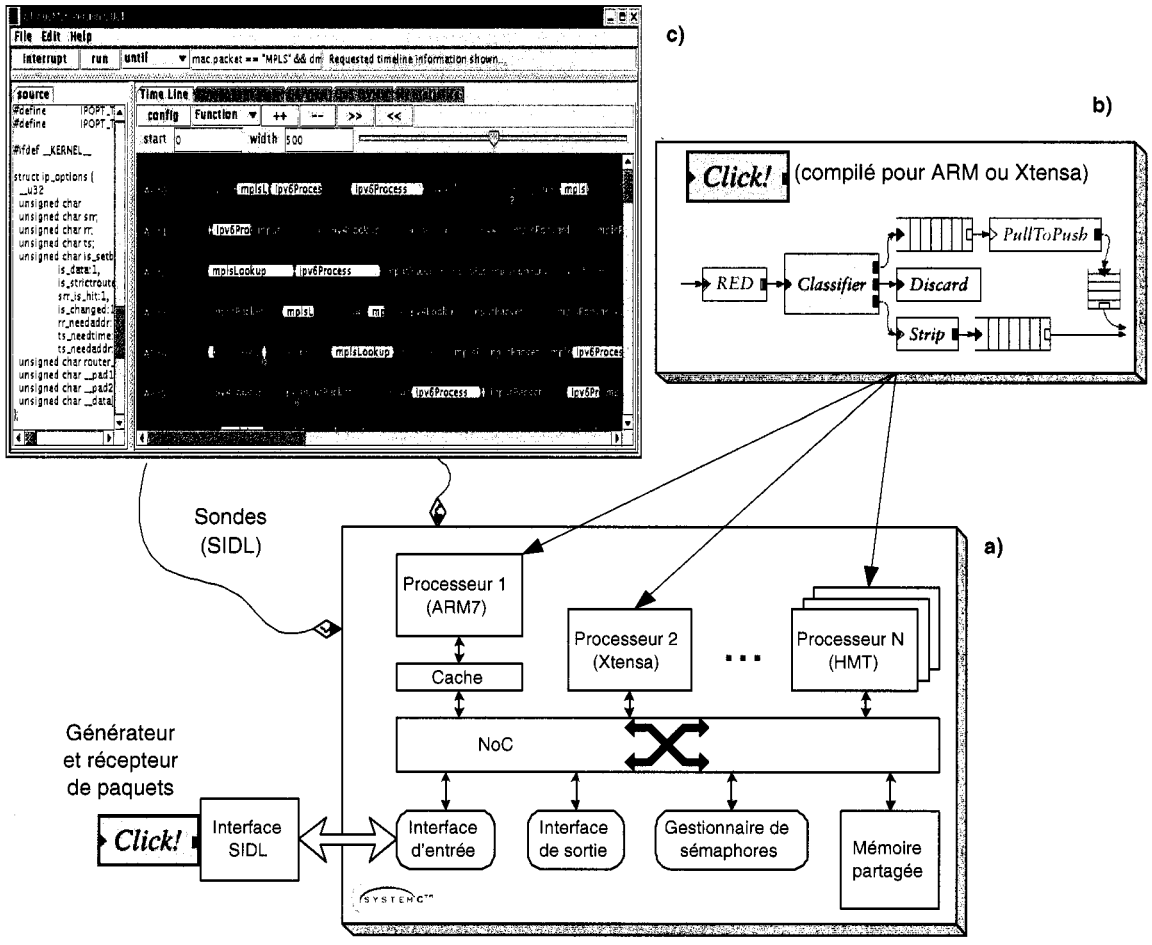


FIGURE 2.1 Les trois parties de la plateforme d'exploration StepNP

l'utilisateur de StepNP en sélectionnant un certain nombre de composants et en les connectant ensemble grâce à leur interface commune. Afin de faciliter la création de cette architecture, un certain nombre de composants usuels sont déjà fournis dans StepNP. Une des convictions de l'équipe derrière StepNP est que les futurs SoC vont principalement utiliser un grand nombre de processeurs embarqués traitant l'information en parallèle avec, au besoin, un minimum de matériel spécialisé afin de répondre aux critères de performances [62]. L'architecture ainsi développée reste flexible puisque la majorité des fonctionnalités sont réalisées en logiciel. C'est pour cette raison que la bibliothèque de composants prêts à être utilisés est surtout composée du nécessaire pour connecter plusieurs processeurs ensemble, c'est-à-dire :

- des modèles de processeurs comme le ARM ou le PowerPC ;
- des modèles fonctionnels de réseau-sur-puce (NoC) et d'interconnexions au niveau BCA ou TF ;
- quelques coprocesseurs comme un gestionnaire de sémaphores ou des modules d'entrées et de sorties ;
- des périphériques de base telles des mémoires partagées et des mémoires caches génériques.

Processeurs. Les modèles de processeurs inclus dans StepNP sont basés sur les ISS du ARM ou du PowerPC disponibles dans le domaine public à travers l'outil de débogage GDB. L'ISS du processeur est encapsulé dans un module SystemC doté de l'interface standard dans StepNP qui est nommée SOCP (voir section 2.3.4). Cela permet au processeur, qui est modélisé avec une précision au niveau des cycles, d'interagir avec le reste du système. L'ISS du processeur ARM est aussi utilisé pour créer un modèle de processeur HMT qui utilise le même jeu d'instructions que le ARM.

interconnexions. StepNP offre plusieurs modèles d'interconnexions qui partagent tous la même interface de type TLM nommée SOCP. Parmi ces modèles nous retrouvons :

- un canal complètement fonctionnel qui ne fait que transférer les requêtes et réponses avec un certain délai configurable ;
- un modèle d'un « cross-bar » fonctionnel où tous les couples maître-esclave possibles peuvent se voir offrir une connexion point à point ;
- un modèle de réseau de type « Mesh 2D » (Hot-Potato par exemple) où les données sont transférées d'un noeud à l'autre sans utiliser de tampons ;
- un NoC nommé « Rotator-on-chip » où tous les composants peuvent s'échanger des messages à travers des tampons rotatifs [18].

Ces modèles sont tous décrits à un haut niveau d'abstraction afin d'offrir un bonne vitesse de simulation et peuvent être raffinés une fois que le choix d'une technique

particulière est arrêté. Au besoin, de nouveaux types d'interconnexions peuvent aussi être développés.

Coprocesseurs. Tel que vu à la section 1.3, les processeurs réseau utilisent des coprocesseurs adaptés à certaines tâches de traitement, telle la recherche dans une table de routage, afin d'obtenir les performances désirées. StepNP offre quelques coprocesseurs génériques dont un gestionnaire de sémaphores, mais il offre surtout une infrastructure pour rapidement développer et intégrer des nouveaux coprocesseurs dans une architecture donnée.

Toutes les composantes de StepNP reposent donc sur SystemC et sont créées à partir de classes de base qui imposent une certaine structure et offrent des fonctionnalités communes. Il suffit d'hériter d'une de ces classes pour créer un nouveau module qui peut s'intégrer dans le reste de la plate-forme. Par exemple, pour créer un coprocesseur qui agit comme esclave dans une plate-forme, il suffit d'hériter de la classe SocpSlaveBase et le nouveau composant possède alors toutes les fonctions nécessaires pour communiquer avec le reste de la plate-forme à travers un canal SOCP.

2.2.3 Plate-forme de développement logiciel : Click

Dans le but de permettre le développement rapide d'une application réseau, StepNP utilise le logiciel Click développé au MIT. Click [45] permet de rapidement construire le logiciel d'un routeur modulaire en assemblant différents éléments de base ensemble. Une description beaucoup plus détaillée de ce qu'est Click et des raisons qui justifient sont utilisation dans StepNP sont disponibles à l'annexe IV. De plus, cette annexe décrit les différents outils de contrôle, d'analyse et de vérification offerts par StepNP et qui sont illustrés à la figure 2.1(c).

2.3 Intégration d'un ISS dans SystemC

À la base, StepNP supporte uniquement les processeurs ARM v.4 et PowerPC (versions 603, 603a et 604). Afin d'atteindre un de nos objectifs qui est de vérifier les gains possibles avec un processeur configurable dans un NPU, il est nécessaire d'ajouter un autre ISS dans StepNP. Pour cette recherche, le choix s'est arrêté sur le Xtensa. Avant d'entrer dans les détails de l'intégration du Xtensa dans StepNP, du modèle de mémoire particulier au Xtensa et de son interface SOCP, la structure générale d'un ISS ainsi que les techniques d'intégration dans SystemC seront présentées.

2.3.1 Structure générale d'un ISS

Un simulateur de jeu d'instructions (ISS) offre un bon compromis entre une simulation purement fonctionnelle du processeur qui n'offre aucun détail de simulation et une simulation très détaillée (niveau RTL) qui demanderait beaucoup trop de temps. L'ISS fournit des informations sur le temps d'exécution (en nombre de cycles), les valeurs des registres, les délais dans le pipeline et les échecs d'accès en mémoire cache. Ce niveau de détail permet donc d'avoir un nombre intéressant d'informations tout en préservant des vitesses de simulation tolérables.

À la base, la structure d'un ISS est relativement simple. Tel qu'illustré à la figure 2.2, le code de l'ISS est constitué d'une boucle sans fin qui lit les instructions binaires une à la fois, les décode et ensuite, selon l'instruction, exécute l'opération désirée. Tous les registres de données et d'états du processeur sont modélisés comme des variables internes à l'ISS. Le code à simuler est habituellement chargé à partir du fichier binaire dans un modèle de mémoire interne à l'ISS au début de la simulation et, par la suite, l'instruction pointée par le compteur de programme (PC) est lue et décodée à chaque cycle. Le PC est mis à jour à chaque tour de la boucle de deux manières possible 1) en l'incrémentant d'une instruction (le cas général) ou 2) en

lui faisant faire un saut lorsque l'instruction est un branchement. La mémoire de données est aussi habituellement modélisée à l'intérieur de l'ISS simplement comme un tableau. Bien que conceptuellement simple à réaliser, le code d'un ISS implique la prise en compte d'un grand nombre de détails ainsi qu'une très bonne connaissance de l'architecture du processeur. Cela se complexifie davantage lorsque les aléas du pipeline et les échecs d'accès en mémoire cache sont aussi modélisés. C'est pourquoi l'ISS d'un processeur donné est généralement fourni par le fabriquant du processeur.

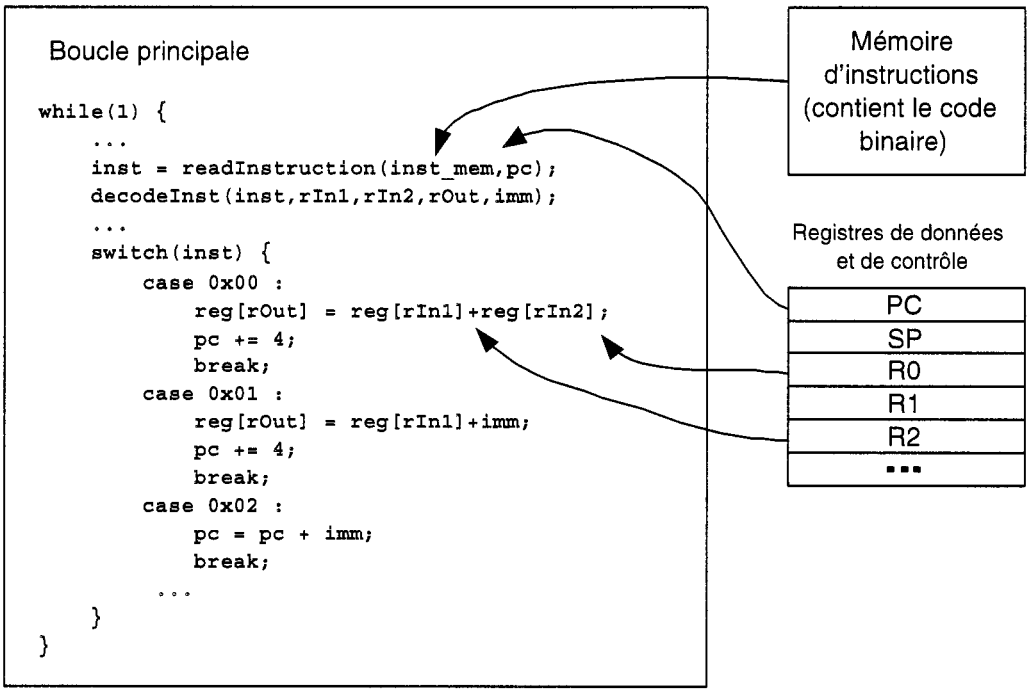


FIGURE 2.2 Structure générale d'un simulateur de jeu d'instructions

Un ISS, tel que fourni par le fabriquant du processeur, est habituellement conçu pour exécuter du code compilé pour le processeur sur un hôte de simulation qui peut avoir une toute autre architecture. Toute l'exécution du code est gérée par l'ISS et il peut arriver que l'ISS fasse appel au système d'exploitation de l'hôte pour certaines fonctions dans le code, comme l'affichage de texte à l'écran. Plusieurs ISS de processeurs commerciaux ne sont pas conçus pour s'exécuter dans un environnement de simulation comme SystemC qui inclut des modules matériels connectés au proces-

seur. Plusieurs des outils de développement présentés à l'annexe III offrent d'ailleurs leurs propres méthodes d'intégration d'un ISS avec un environnement de simulation matérielle pour compenser cette lacune.

Il existe trois méthodes de base, illustrées à la figure 2.3, pour intégrer un ISS dans SystemC [8].

a) L'ISS et la plate-forme SystemC roulent dans deux processus différents et un mécanisme de communication assure la synchronisation et l'échange de données entre les deux mondes.

b) L'ISS est accédé grâce à un mécanisme de communication déjà existant, mais le tout se fait à l'intérieur d'un module SystemC qui enveloppe l'ISS et traduit ses requêtes externes en appels TLM sur le canal SystemC.

c) Le code original de l'ISS est modifié et directement intégré dans un module SystemC avec les interfaces requises.

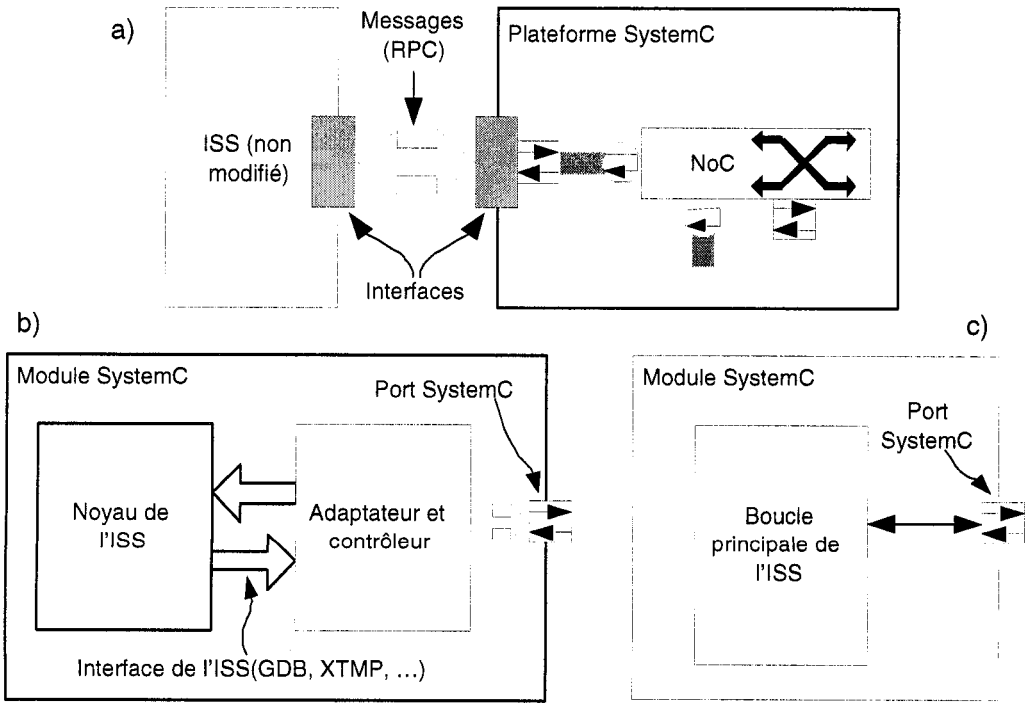


FIGURE 2.3 Trois différentes techniques pour intégrer un ISS dans SystemC

La première approche servant à effectuer une co-simulation entre un ISS et StepNP est donc d'utiliser une interface (comme SIDL, voir annexe IV.3.2) qui permet d'échanger des messages entre l'ISS et SystemC, tel qu'illustrée à la figure 2.3(a). Pour que cette approche fonctionne, le canal de communication dans la simulation SystemC doit posséder une interface agissant comme un pont capable de parler au processeur et au reste de la plate-forme. Cette approche est pratique puisqu'elle ne demande aucune modification à l'ISS. Malheureusement, l'interface entre l'ISS et SysyemC peut s'avérer complexe. En effet, connecter deux simulateurs différents ensemble, tel que réalisé par Seamless CVE de Mentor Graphics (avec VHDL et un ISS), n'est pas nécessairement une tâche aisée. Le principal désavantage reste cependant les vitesses de simulation qui ne sont pas très reluisantes si le processeur échange beaucoup de données avec le reste de la plate-forme. Avec ce type de connections, il n'est donc pas recommandé que la mémoire d'instruction soit simulée dans la plate-forme. Elle doit être inclue dans le code de l'ISS.

La seconde technique, illustrée à la figure 2.3(b), se base sur le fait que plusieurs ISS sont pourvus d'une interface qui permet de contrôler dynamiquement leur exécution. Par exemple, plusieurs ISS sont fournis avec le code source de l'outil de débogage GDB. GDB utilise ces ISS pour permettre de déboguer du code d'un processeur en se servant d'un autre type de processeur comme hôte pour la simulation. Afin de contrôler la simulation, GDB utilise une interface générique qui permet de faire avancer la simulation pour un certain nombre de cycles (ou jusqu'à ce que l'exécution soit complétée), d'insérer des points d'arrêt dans le code et d'aller lire les valeurs des registres du processeur. Il est possible de créer un module SystemC qui utilise cette interface afin de contrôler l'ISS [23, 8]. Cette approche est aussi utilisée dans StepNP et son principal avantage est qu'avec un seul adaptateur SystemC à GDB, il est possible de supporter un bon nombre de modèles de processeurs offert par GDB.

La troisième approche, illustrée à la figure 2.3(c), utilise le code source de l'ISS afin d'en extraire le noyau et de l'intégrer directement dans un module SystemC. Le

simulateur SystemC se retrouve alors à directement contrôler la boucle principale de l'ISS qui peut, par exemple, être exécutée une fois par cycle d'horloge. L'avantage de procéder ainsi est qu'un niveau d'indirection (l'API de GDB par exemple) se trouve éliminé, ce qui augmente les vitesses de simulation. Le désavantage est que le code source de l'ISS doit être disponible, ce qui n'est pas toujours le cas, et que si nous désirons supporter un second processeur, une partie du travail d'intégration est à refaire.

Les approches b) et c) ont quelques points en commun. Tout d'abord, pour ces cas, on ne parle plus vraiment de co-simulation entre le logiciel et le matériel, mais bien d'une simple simulation puisque le même langage (C/C++) et le même simulateur (SystemC) sont utilisés pour simuler toutes les parties de la plate-forme. Cela facilite grandement la synchronisation entre le logiciel et le matériel et élimine une bonne partie de la complexité reliée à la synchronisation entre le logiciel et le matériel que des programmes comme Seamless résolvent. De plus, avec ces deux techniques, il peut être intéressant d'extraire les modèles de mémoire de l'ISS pour les placer sur un canal de communication partagée dans la plate-forme SystemC. Cela permet d'avoir des mémoires partagées entre processeurs et coprocesseurs et aussi d'obtenir des estimés de performance plus réalistes qu'avec les mémoires à accès instantanés que l'on retrouve typiquement dans un ISS. Pour extraire ces mémoires, il suffit de rediriger tous les accès mémoire de l'ISS vers le port de communication SystemC du module.

Une autre question à se poser lorsque l'on désire contrôler un ISS dans SystemC concerne le niveau désiré de granularité des interactions entre l'ISS et le reste du système. Deux différents facteurs affectent cette granularité : l'emplacement de la mémoire et les moments de synchronisation avec le reste de la plate-forme. Tel que mentionné précédemment, une mémoire d'instructions et de données internes à l'ISS se simule beaucoup plus rapidement, mais offre moins de mesures de performances. Le moment où l'ISS se synchronise avec SystemC et le reste de la plate-forme influence

aussi le temps de simulation. La première solution est de contrôler la boucle principale de l'ISS avec une horloge SystemC, c'est-à-dire qu'à chaque cycle d'horloge, l'ISS lance une nouvelle instruction. Cette manière de procéder offre une précision au niveau des cycles et permet d'obtenir des traces de simulation précises, mais elle peut entraîner de longs délais. En effet, un processeur demande typiquement un très grand nombre d'instructions (donc de cycles) pour réaliser une fonction si on le compare à un coprocesseur matériel. Habituellement, le processeur doit communiquer avec le reste de la plate-forme uniquement lorsque le calcul est fini (si l'on considère qu'il dispose de mémoires cache locales). De bons gains peuvent donc être obtenus en laissant l'ISS rouler librement et parallèlement au reste de la plate-forme jusqu'au moment où il requiert un accès externe. À ce moment, il faut synchroniser les deux mondes en bloquant le processeur jusqu'à ce que sa requête soit traitée par le module matériel ciblé. Il s'agit là de la technique généralement employée avec l'ISS de la figure 2.3(a). Cela demande un adaptateur SystemC pour l'ISS plus complexe et fausse alors les résultats de profilage. Mais les gains en terme de vitesse de simulation sont importants. Le modèle du processeur ARM dans StepNP peut fonctionner selon les deux modes d'opération.

2.3.2 Intégration du Xtensa dans StepNP

Tensilica offre un ISS pour son processeur Xtensa qui a la particularité de pouvoir s'adapter aux différentes configurations possibles du Xtensa, y compris les instructions TIE personnalisées. Lorsqu'une configuration est réalisée sur le site Internet de Tensilica, l'usager peut alors télécharger les fichiers relatifs à cette configuration. Parmi ceux-ci se trouve tout le nécessaire pour configurer l'ISS de base. Tensilica fournit un simulateur exécutable de base auquel on doit fournir un code compilé ainsi que le nom de la configuration désirée. Ce simulateur est aussi capable de modéliser la mémoire cache ainsi que les aléas du pipeline et il peut générer un fichier de statistiques en cours de simulation afin d'obtenir des résultats de profilage détaillés à la

fin de la simulation. Le tableau 2.1 dresse la liste des outils logiciels fournis avec le Xtensa.

TABLEAU 2.1 Les outils logiciels fournis avec le Xtensa

Outil	Description	Exécutable
Simulateur (ISS)	ISS pour un seul processeur	xt-run
Compilateur GNU	Compilateur GCC de GNU adapté au Xtensa	xt-gcc, xt-g++
Compilateur Xtensa	Compilateur C/C++ de Xtensa	xcc
Profileur	Outil de profilage de GNU adapté au Xtensa	xt-gprof, xt-trace
Debugeur	Outil de débogage de GNU adapté au Xtensa	xt-gdb, xt-ddd
API vers l'ISS	Interface de programmation pour créer son propre ISS	XTMP

L'ISS de base fournit par Tensilica est limité à simuler un seul processeur sans aucun périphérique autre qu'une mémoire. Lors de la création d'un SoC, cela s'avère nettement insuffisant, une interface de programmation (API) vers le noyau de l'ISS est donc fournie. Cette interface se nomme XTMP (*Xtensa Modeling Protocol*) et offre le seul et unique moyen d'accéder à l'ISS puisque le code source de ce dernier n'est pas disponible ; ce qui est fort malheureux. Face à la non disponibilité du code source de l'ISS nous nous sommes donc tournés vers la méthode de la figure 2.3(b) pour intégrer l'ISS du Xtensa dans StepNP. Le tableau 2.2 dresse une liste partielle des fonctions disponibles avec l'interface de programmation XTMP. Grâce à XTMP il est possible de démarrer et d'arrêter la simulation pour ainsi aller chercher les valeurs des registres comme c'est le cas avec l'interface de GDB. Cependant XTMP possède une différence importante par rapport aux ISS inclus dans GDB : il est capable de créer des simulations impliquant plusieurs processeurs. Avec l'ISS du ARM utilisé dans StepNP, il a été nécessaire de modifier ce dernier afin que plus d'une instance de l'ISS puisse être intégrée dans une simulation. De plus, XTMP offre tout le nécessaire pour modéliser des coprocesseurs matériels qui échangent des données avec les Xtensa

ainsi que d'autres constructions utiles dans un environnement multiprocesseur tels des mutex et sémaphores (XTMP_lock).

TABLEAU 2.2 L'interface de programmation XTMP

Nom	Fonction
XTMP_coreNew()	Crée un nouveau processeur
XTMP_getRegisterValue()	Retour la valeur d'un registre
XTMP_start()	Début l'exécution pour X cycles
XTMP_setInterrupt()	Active ou désactive une intéruption
XTMP_stop()	Arrête la simulation
XTMP_connect()	Connecte un objet XTMP_device au CPU
XTMP_wait()	Pause la simulation pour X cycles
XTMP_deviceNew()	Crée un object XTMP_device
XTMP_clockTime()	Retourne le temps vu par le CPU
XTMP_sysRamNew()	Crée une mémoire interne à l'ISS
XTMP_loadProgram()	Charge un fichier en mémoire
XTMP_enableDebug()	Permet le debugage sur un CPU donné

Toutes ces fonctionalités offertes par XTMP permettent de s'en servir comme moteur de la simulation. En effet, la philosophie derrière XTMP est que ce dernier contrôle complètement la simulation et toutes les interactions à l'intérieur de la plate-forme se font au travers d'appels de fonction XTMP. Il est possible de créer des coprocesseurs en SystemC, mais ces derniers doivent, toujours selon la philosophie d'utilisation de XTMP, être inclus dans un object XTMP_device qui possède une interface transactionnelle précise et qui contrôle l'horloge. Les modèles de mémoire interne à l'ISS utilisent aussi cette interface pour communiquer avec l'ISS. L'interface d'un XTMP_device est donnée dans le tableau 2.3. La dernière colonne du tableau indique s'il s'agit d'une fonction qui modélise un type d'accès réel du processeur ou si la fonction est uniquement utilisée en simulation. Par exemple, la fonction `bcopyFromHost` est utilisée pour copier le fichier binaire de l'application en mémoire avant le début de la simulation ; la fonction `bcopyToHost` est utilisée de son côté pour envoyer une chaîne de caractères à l'hôte de la simulation afin qu'il affiche les écritures à l'écran (ceci est utilisé pour simuler un `printf()` par exemple). Ces deux fonctions sont

uniquement des artifices de simulation.

TABLEAU 2.3 L'interface d'un module générique dans l'environnement XTMP

Fonction	Description	Réel
read()	Lecture d'un mot	Oui
write()	Écriture d'un mot	Oui
blockRead()	Lecture d'un bloc de mots (burst read)	Oui
blockWrite()	Écriture d'un bloc de mots	Oui
peek()	Lecture d'un mot sans délais, utile pour le débogage	Non
poke()	Écritue d'un mot sans délais	Non
bcopyFromHost()	Transfert d'une série d'octets de l'hôte de la simulation à l'ISS	Non
bcopyToHost()	Transfert d'une série d'octets de l'ISS à l'hôte	Non
ticker()	Fonction sensible à l'horloge, automatiquement appelée par XTMP	-
writeBusyTime	Délais générés par une opération d'écriture	-
userdata	Champ pouvant contenir une donnée ou un pointeur défini par l'usager	-

Lorsque l'on désire intégrer l'ISS du Xtensa dans SystemC (afin de l'utiliser dans StepNP par la suite), toutes ces fonctionnalités offertes par XTMP causent malheureusement un problème. Il devient nécessaire d'utiliser la bibliothèque XTMP non pas pour contrôler la simulation, mais bien à l'intérieur d'un module SystemC qui devient le maître de la simulation. Pour ce faire, la bibliothèque XTMP doit être utilisée d'une manière non prévue initialement. La documentation disponible sur XTMP, qui explique comment se servir de cet outil et non son fonctionnement interne, devient alors bien moins utile. Lors de la création d'un module SystemC pour le Xtensa deux grands objectifs doivent être atteints : 1) la communication entre l'ISS et le reste de la plate-forme SystemC doit être facile à réaliser et 2) il doit être possible de synchroniser l'ISS avec le simulateur SystemC.

Dans un premier temps, il est donc nécessaire de rediriger les appels de l'ISS vers les mémoires RAM ou ROM dans le module SystemC qui englobe le processeur. Cela

permet, tel qu'illustré à la figure 2.4, de rediriger ces appels soit vers un modèle de mémoire interne au module, ou encore sur le canal transactionnel SystemC (nommé SOCP). Deuxièmement, la bibliothèque XTMP requiert une interface vers un système de traitement multiprocessus tels les pthread de POSIX. Ce pilote permet à chaque processeur de rouler dans son processus indépendant (peu importe s'il y a un ou plusieurs processeurs dans la simulation) et facilite la synchronisation entre les processeurs. Le système multiprocessus n'a pas besoin d'être préemptif, il est donc possible d'utiliser des bibliothèques plus légères que POSIX tels QuickThread [43] ou encore SystemC. Par défaut, XTMP fournit un pilote permettant d'utiliser QuickThread ou SystemC. Notons que SystemC utilise lui-même QuickThread pour simuler les `SC_THREAD` et autres processus lorsqu'il est compilé sur une machine Unix. La principale différence entre QuickThread et SystemC est donc que SystemC offre une interface plus simple vers QuickThread. Le module SystemC qui englobe le Xtensa doit donc implémenter les fonctions pilotes requises par XTMP afin de contrôler la simulation et de faciliter l'intégration avec le reste de la plate-forme. Les fonctions du pilote ainsi que leur implémentation en SystemC sont décrites dans le tableau 2.4. Notons finalement que XTMP n'offre aucun mécanisme pour simuler la mémoire cache à l'extérieur de l'ISS. La seule approche possible, puisque le code source de XTMP n'est pas disponible, est de créer une configuration d'un Xtensa sans cache et d'implémenter sa propre cache au niveau du décodeur d'adresses. Malheureusement cela réduit grandement les vitesses de simulation tel que nous le verrons plus loin au chapitre 4.

Nous obtenons ainsi un module SystemC qui englobe l'ISS du Xtensa et qui est capable de communiquer avec un canal TLM ainsi qu'avec des mémoires partagées décrites en SystemC. Ce module SystemC possède un port d'horloge et exécute une instruction à chaque cycle. Les deux sous-sections suivantes vont s'attarder un peu plus sur le modèle de mémoire utilisé par le Xtensa ainsi que sur le canal de communication SOCP.

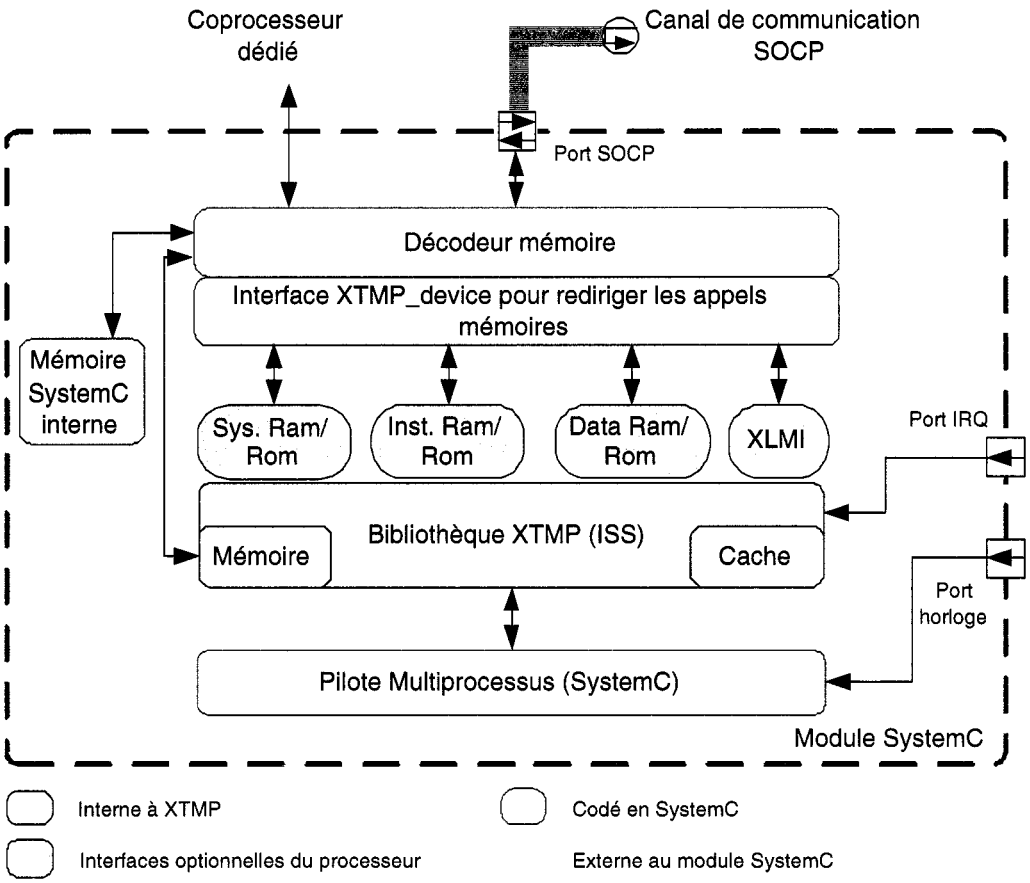


FIGURE 2.4 Le simulateur du Xtensa intégré dans StepNP

2.3.3 Modèle de mémoires

Comme nous l'avons spécifié précédemment, il peut être utile dans certains cas de sortir le modèle de mémoire à l'extérieur de l'ISS lors de la simulation. Cela peut permettre de modéliser plusieurs architectures qui ne seraient pas bien représentées si l'ISS possédait une mémoire inaccessible à partir de la plate-forme SystemC, par exemple :

- une mémoire d'instruction partagée par plusieurs processeurs ;
- une mémoire partagée avec un coprocesseur pour échanger des données ;
- une mémoire accessible uniquement au travers d'un canal de communication qui possède une latence non négligeable.

TABLEAU 2.4 L'interface du pilote multiprocessus de XTMP implémenté en SystemC

Fonction du pilote	Description des fonctions	Implémentation en SystemC
threadNew()	Création d'un nouveau module matériel (un coprocesseur par exemple)	SC_THREAD()
cpuThreadNew()	Création d'un nouveau processus contenant un processeur	SC_THREAD()
driverStart()	Début de la simulation, lance tous les processus	sc_start()
driverStop()	Arrête la simulation	sc_stop()
driverWait()	Un processus se place en attente, force un changement de contexte	wait(t, unitée)
clockTime()	Retourne à l'ISS le temps vu par la plate-forme simulée	sc_time_stamp() ou sc_simulation_time()

Un modèle de mémoire décrit en SystemC a donc été réalisé. Cette mémoire peut être connectée à l'ISS à l'intérieur même du module SystemC du processeur (figure 2.4) ou encore elle peut être connectée sur un canal SOCP. Un modèle de mémoire simple est trivial à réaliser en SystemC. Il suffit de créer un module avec une interface transactionnelle pour les requêtes de lecture et d'écriture et de modéliser la mémoire comme un tableau en C++. Les adresses correspondent alors à un index dans le tableau. Bien entendu, plusieurs optimisations peuvent être apportées à ce modèle simpliste. Par exemple, nous pouvons utiliser un mécanisme d'allocation plus évolué pour le tableau C++ afin de modéliser une mémoire, où une plage d'adresses est allouée uniquement lorsqu'elle est utilisée pour la première fois. Cette modélisation d'une mémoire paginée permet de sauver de l'espace mémoire sur la machine hôte de la simulation. Dans le cas d'un modèle de mémoire pouvant se connecter au Xtensa ou au canal SOCP, il est nécessaire de supporter l'interface d'un XTMP_device (tableau 2.3) ainsi que l'interface SOCP (section 2.3.4). Cela n'est pas un problème en soi. Le défi provient plutôt de la nature configurable du Xtensa. En effet, tel que présenté dans le tableau 1.1, le processeur peut avoir une interface de 32, 64 ou 128

bits avec la mémoire et il peut fonctionner sous le mode « big-endian » ou « little-endian ». Il est donc nécessaire de créer un modèle de mémoire qui peut s'adapter automatiquement à ces différentes configurations. Par exemple, un processeur big-endian avec une interface de 128 bits qui effectue une lecture en bloc de 32 octets, s'attend à recevoir les mots dans l'ordre inverse d'un processeur little-endian.

2.3.4 Canal SOCP

Un canal SOCP (*SystemC Open Core Protocol*), dont nous avons parlé à plusieurs reprises dans les sous-sections précédentes, est un canal de communication TLM. C'est-à-dire qu'il implémente une interface transactionnelle permettant aux maîtres et aux esclaves d'une plate-forme d'échanger des messages selon un format prédéfini et flexible. Ce format mimique le protocole *Open Core Protocol* (OCP) tel que spécifié par l'association OCP-IP [55]. OCP définit une interface et un protocole que tous les modules de la plate-forme doivent supporter afin d'être compatibles et être capable de s'échanger des messages. OCP ne définit pas un modèle physique d'interconnexions qui permet d'acheminer ces messages. OCP est une version étendue du protocole VCI de VSIA [17] et supporte, entre autres :

- un identificateur unique pour chaque maître et esclave du système (`mConnID`) ;
- un identificateur unique pour chaque processus d'un processeur ou coprocesseur HMT (`mThreadID`) ;
- des requêtes concurrentes et des réponses dans le désordre (*split transactions*) supportées à l'aide des identificateurs uniques ;
- des lectures et écritures en rafale (*burst access*).

StepNP définit sa propre interface SOCP comme une version simplifiée, mais supportant la grande majorité des fonctionnalités, de OCP. Cette interface est définie en détail dans [60] et est composée d'une structure en C qui comprend tous les champs nécessaires à la communication dont les identificateurs uniques, la taille de la requête, la taille de la réponse, des champs de statut et, bien sur, les données. Notons qu'une

version officielle et un peu plus complexe d'un protocole OCP décrit en SystemC a été réalisée par le groupe OSCI [31], mais puisque cette version n'est pas supportée dans StepNP, elle ne sera pas décrite ici.

La figure 2.5 illustre les concepts de base d'une communication TLM avec SOCP. Les maîtres possèdent un port qui se connecte sur une interface de type maître implémentant la fonction `putReq()` qui sert à envoyer une requête. De leur côté, les esclaves ont un port qui se connecte sur une interface de type esclave implémentant la fonction `putRsp()` qui permet d'envoyer la réponse à la requête. De plus, le maître implémente l'interface esclave et vice-versa. Cela permet de connecter directement un maître et un esclave ensemble (figure 2.5(a)) ou encore de passer par un canal intermédiaire (figure 2.5(b)). Aucune restriction n'est imposée sur le type de canal, il peut s'agir d'un modèle purement fonctionnel qui ne fait que passer les messages aux bons destinataires sans délai comme d'un modèle beaucoup plus détaillé (i.e. un modèle au niveau BCA). Bien entendu le canal peut posséder une multitude de ports maîtres et esclaves afin de supporter un grand nombre de composants dans la plate-forme.

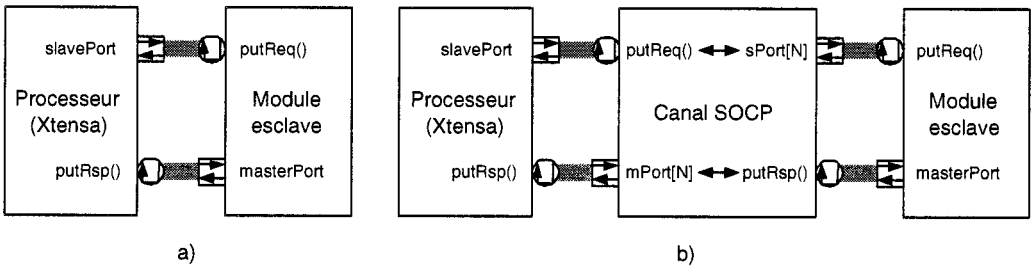


FIGURE 2.5 L'interface de communication transactionnelle SOCP

Tel qu'illustré à la figure 2.4 le Xtensa possède plusieurs interfaces mémoires configurables et le décodeur mémoire du processeur (implémenté dans notre cas à même le module SystemC) est chargé de rediriger les appels de l'ISS vers la bonne mémoire. La largeur des mots échangés avec ces mémoires ainsi que la plage d'adresses utilisée par chacune d'entre elle est bien entendu configurable. De plus, certaines de ces interfaces sont optionnelles et peuvent ne pas être utilisées sur une configuration du processeur.

Le tableau 2.5 décrit les cinq (5) types d'interfaces mémoires disponibles et indique s'il est nécessaire que cette mémoire soit locale au processeur ou non. Le module SystemC enveloppant l'ISS détecte automatiquement de quelle interface provient l'appel mémoire. Il est donc possible de rediriger les appels soit dans une mémoire locale (interne au module SystemC du processeur) ou vers le canal SOCP. Le type d'interface utilisé par le Xtensa ou encore selon une plage d'adresses peuvent être utilisés pour prendre cette décision. Les appels fait à la mémoire système peuvent être redirigés vers l'une ou l'autre des destinations sans problème. Il en va de même pour le port XLMI (*Xtensa Loacal Memory Interface*) puisqu'il supporte un mode bloquant où le processeur doit attendre après la réponse. Bien qu'il soit conçu pour être connecté à un coprocesseur local et non derrière un canal de communication partagé, il se peut en effet que le coprocesseur réponde après un long délai au Xtensa. Cependant les interfaces vers les mémoires locales de données et d'instructions possèdent des contraintes strictes au niveau des modes d'accès et des temps de réponse. Il n'est donc pas recommandé de rediriger les appels vers ses derniers sur le canal SOCP qui risque de posséder des délais non négligeables et non prévisibles.

TABLEAU 2.5 Les différents type d'interface mémoire du processeur Xtensa

Nom	Description	Local
System RAM/ROM	Mémoire ou bus accessible au travers du PIF (processor interface)	Non
Data RAM/ROM	Mémoire synchrone avec le processeur, temps d'accès garanti	Oui
Instruction RAM/ROM	Mémoire synchrone avec le processeur, temps d'accès garanti	Oui
XLMI	Permet d'échanger des données avec un périphérique accessible comme une mémoire	Non
Cache	Mémoire cache du processeur	Oui

Finalement, notons que par défaut, le Xtensa ne générera aucun appel de lecture ou d'écriture en bloc sur l'interface XLMI. Si cette dernière est branchée sur le canal SOCP, une multitude d'appels sur le canal seront générés ce qui n'est pas la manière

la plus efficace de communiquer une grande quantité d'informations. Si l'on désire remédier à ce problème, il est facile de placer un tampon entre le processeur et le canal qui accumule une certaine quantité de données avant de les envoyer sur le canal et qui, au retour, offre le résultat mot par mot à l'interface XLMI.

2.4 Méthodologie de codesign employée

Avec SystemC, StepNP, Click et le Xtensa, nous disposons de tous les outils nécessaires pour effectuer une exploration architecturale d'un processeur réseau utilisant des instructions spécialisées. La création de ce type de SoC reste un problème de codesign typique et bien qu'il existe plusieurs voies, souvent reliées aux outils utilisés, pour réaliser le circuit, l'approche générale reste sensiblement la même. Cette approche classique est présentée à la figure 2.6 (boîtes blanches).

Le flot de conception débute par une spécification du produit à concevoir. Les objectifs, les fonctionnalités et les contraintes visés sont alors décrites. Plusieurs langages, formels ou non, spécialisés pour la spécification des requis existent, un des plus populaires étant toujours le logiciel de traitement de texte. A partir de ce point, il est essentiel de rapidement avoir un design précis et vérifiable, qui offre beaucoup plus de détails qu'une spécification souvent trop imprécise. Pour ce faire, différents langages de haut niveau, tels C++ et SystemC, peuvent être utilisés pour développer un modèle fonctionnel et simulable sans notion de temps [14].

La seconde étape consiste à profiler l'application initiale afin d'obtenir des premières mesures de performance. Le profilage s'effectue en exécutant l'application avec plusieurs stimulis afin de déterminer quelles sont les portions qui demandent le plus de temps d'exécution. Ces premières simulations permettent aussi de valider si l'application est fonctionnelle et si elle donne les bons résultats. Afin d'obtenir un SoC flexible et développé à faibles coûts, la portion de l'application qui roule sur des pro-

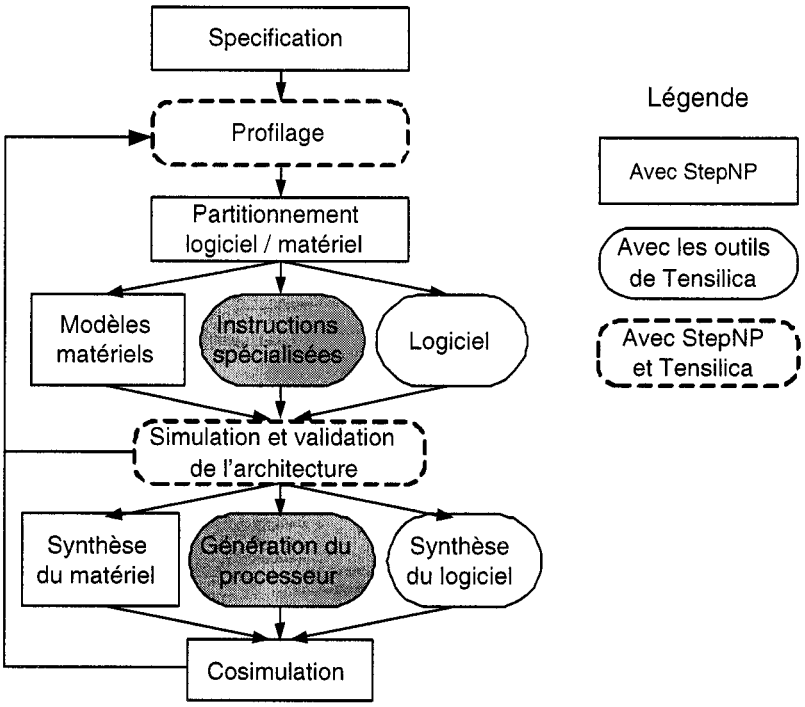


FIGURE 2.6 Méthodologie de co-design traditionnelle (carrés blanc) et les modifications proposées (carrés ombrés)

cesseurs doit être maximisée. L'avantage offert par le logiciel est dû à la facilité de développer du logiciel vis-à-vis du matériel dédié et aussi au fait que plusieurs processeurs pouvant facilement être intégrés dans un SoC sont déjà disponibles. C'est pour ces raisons qu'il est utile d'avoir, dès la phase de spécification, une version purement logicielle de l'application. Cette version servira non seulement de modèle de référence pour valider les fonctionnalités de la plate-forme finale, mais aussi de point de départ pour le raffinement. La première étape de raffinement vise donc à optimiser les portions de l'application qui, à la lumière du premier profilage, demandent trop de cycles pour s'exécuter. Ces optimisations logicielles demandent généralement une connaissance de l'architecture du processeur qui sera utilisé. En plus de retravailler certaines boucles critiques, une autre technique efficace consiste à paralléliser l'application afin qu'elle s'exécute sur plusieurs processeurs où sur un processeur HMT.

L'optimisation du logiciel est un processus itératif où plusieurs profilages différents

sont réalisés. Il se peut que les contraintes souhaitées soient déjà respectées après cette étape, ce qui est idéal et donne ainsi un système relativement simple à réaliser. Cependant dans le cas d'applications qui traitent un haut débit de données, telle une application réseau ou multimédia, une solution purement logicielle arrivera rarement aux performances désirées. À ce moment, l'étape de partitionnement logiciel/matériel entre en jeu. Les portions de l'application qui ralentissent l'exécution peuvent se voir transférer vers un module matériel dédié. Ce dernier peut grandement accélérer l'exécution en exploitant le maximum de parallélisme possible ou encore en possédant des unités arithmétiques capables d'effectuer des manipulations de bits non standards. Afin d'accélérer le développement de ces modules matériels, il peut être utile d'essayer de réutiliser des blocs déjà existants, d'où la grande utilité de protocoles de communication tel OCP qui permettent l'intégration rapide de différents blocs matériels. Cette recherche architecturale, qui est formée des étapes de profilage, de partitionnement et de simulation, demande souvent plusieurs itérations avant de converger vers une solution acceptable. Dans ce contexte, un environnement de développement comme StepNP s'avère très utile puisqu'il permet de rapidement créer et modifier à haut niveau d'abstraction, une architecture donnée. De plus, il est nécessaire de pouvoir raffiner la plate-forme à un niveau suffisant pour pouvoir être confiant que les résultats de profilage obtenus sont significatifs. Pour ce faire, des délais sont ajoutés aux différents composants de la plate-forme et certains modules, tel le canal TLM, peuvent être remplacés par un modèle plus détaillé. Dans tous les cas, l'exploration architecturale demeure une activité complexe principalement à cause de l'énorme quantité de solutions possibles. L'expérience du concepteur reste un des atouts les plus précieux. L'article [26] présente de manière assez exhaustive l'ensemble des différentes stratégies d'exploration architecturale.

Les dernières étapes se penchent sur le raffinement des modules matériels vers une représentation synthétisable ainsi que sur la compilation du logiciel sur les processeurs choisis. Cette étape est nécessaire puisque les délais utilisés lors du partitionnement

sont habituellement des estimés. Ces étapes de raffinement impliquent aussi de raffiner les communications TLM vers des modèles précis au niveau des broches et des cycles d'horloge. Il est aussi souvent nécessaire de générer les interfaces de communication des modules matériels ou d'adapter des interfaces déjà existantes au protocole de communication sélectionnée [13]. Puisque dans ce travail nous nous intéressons principalement à l'exploration architecturale, ces dernières étapes ne seront pas approfondies.

Finalement, notons que cette méthodologie s'applique au développement d'une nouvelle plate-forme. Tel qu'expliqué dans l'introduction, l'approche la plus simple pour développer un nouvel SoC est de réutiliser une plate-forme existante et de l'adapter aux nouveaux besoins en autant que les nouveaux requis ne soient pas radicalement différents des anciens. Cela peut se faire lorsque l'ancienne et la nouvelle plate-forme travaillent sur la même classe d'application, tel le traitement de paquets. Cette approche est souvent utilisée dans l'industrie.

2.4.1 Différents types de processeurs

Lors de l'étape de partitionnement logiciel/matériel, une décision importante concernant le type de processeur utilisé doit être prise. Même si la création d'un nouveau processeur maison n'est habituellement pas considérée, étant donnée la complexité et le temps requis par une telle tâche, le choix qui s'offre au développeur reste vaste. Il est, en effet, possible de choisir parmi plusieurs technologies présentées à la figure 2.7. Cette figure indique aussi le compromis entre la performance et la flexibilité impliqué par chacune des différentes technologies.

À une extrémité du spectre se trouve le processeur général. Plusieurs processeurs embarqués bien connus font parties de cette catégorie dont les ARM, MIPS et PowerPC. Ces processeurs possèdent généralement une architecture fixe de type RISC capable

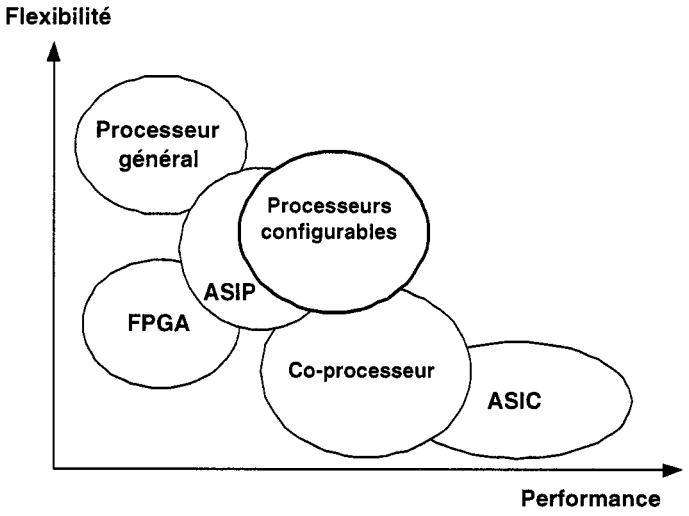


FIGURE 2.7 Spectre des différentes technologies de processeurs et leurs compromis au niveau de la performance et de la flexibilité

de s'acquitter d'un grand nombre de tâches, d'où sa flexibilité. Cependant pour des applications demandant de hautes performances, comme le traitement réseau, il s'avère insuffisant et son architecture fixe empêche toute optimisation. C'est pourquoi le processeur général est souvent couplé à du matériel plus dédié, comme le coprocesseur qui se situe à l'autre extrémité du spectre sur le graphique. Le matériel dédié offre, en effet, les meilleures performances possibles, mais il est long à développer et, un fois réalisé, ne se prête pas très bien aux changements.

Bien que la performance des processeurs d'usage général ne cesse d'augmenter, leur efficacité reste limité pour l'exécution de certaines applications embarquées. C'est pour cette raison que le choix d'un ASIP (*Application-Specific Instruction-set Processors*) peut s'avérer avantageux. Ce type de processeur concède un peu de flexibilité au profit d'un jeu d'instructions et d'unités fonctionnelles spécialisées à une certaine classe d'application. Par exemple, un DSP, qui peut être considéré comme un ASIP, va typiquement inclure une instruction réalisant la multiplication de deux nombres suivie d'une addition ce qui permet, entre autre, d'implémenter un filtre FIR souvent utilisé en traitement de signal. Les unités fonctionnelles spécialisées des ASIP restent

cependant assez simple et ne peuvent pas réaliser d'opérations complexes.

Au milieu du spectre, se trouvent les processeurs configurables qui ont été présentés à la section 1.1. Cette solution offre un compromis intéressant entre la performance et la flexibilité et sa popularité est grandissante [19, 32]. Le concepteur peut facilement utiliser cette technologie pour ajouter des unités fonctionnelles spécifiques à l'application en plus de choisir plusieurs options architecturales comme la dimension des bus et le type de mémoire cache afin d'augmenter les performances de l'application. À l'aide de Click (pour développer le logiciel), de StepNP (pour créer la plate-forme) et du Xtensa, la présente recherche explorera les possibilités offertes par une telle technologie.

2.4.2 Modifications apportées à la méthodologie

Afin de tirer avantage des processeurs configurables, une modification à l'approche de codesign classique est proposée. Cette modification est illustrée par les boîtes ombrées de la figure 2.6. Elle tend à favoriser l'utilisation du processeur configurable comme un compromis aux architectures classiques qui utilisent les deux extrémités du spectre des solutions présentées à la figure 2.7. Plus précisément la phase d'exploration architecturale se divise maintenant en quatre(4) grandes étapes :

1. **Réalisation du logiciel.** Avant d'investiguer d'autres solutions, l'application est complètement réalisée en logiciel afin de débuter l'exploration architecturale sur une base concrète et aussi de maximiser l'utilisation du logiciel dans la plate-forme finale.
2. **Choix et configuration des processeurs.** Le ou les processeurs configurables appropriés sont choisis et les options architecturales appropriées sont sélectionnées.
3. **Instructions spécialisées.** Avant de se lancer dans la création ou la réutilisation de matériel dédié, il est avantageux d'investiguer les possibilités

de créer des instructions spécialisées pour optimiser les boucles critiques de l'application. Ces instructions ont l'avantage d'offrir des gains de performances intéressants tout en étant rapide à créer et surtout facile à intégrer dans le processeur (surtout avec une technologie mature comme celle de Tensilica).

4. **Matériel dédié.** Dans certains cas, il peut être plus avantageux d'utiliser un coprocesseur dédié pour accélérer le traitement. Cela survient lorsque les instructions spécialisées ne sont pas en mesure d'apporter des gains intéressants ou encore lorsque l'on désire avoir une unité fonctionnelle complètement découplée du processeur et qui peut être partagée.

CHAPITRE 3

OPTIMISATION DU JEU D'INSTRUCTION D'UN PROCESSEUR RÉSEAU

Deux applications concrètes basées sur les processeurs réseau de la méthodologie présentée à la section 2.4 sont réalisées dans ce chapitre. Ces applications permettront d'évaluer les différents avantages que peuvent apporter les processeurs configurables. Les deux applications sont réalisées avec Click et implémentent des fonctionnalités typiquement réalisées par des routeurs. La première application, l'algorithme de routage des paquets IPv4, est grandement utilisée à travers tout le réseau Internet pour acheminer les paquets à destination. Cette tâche est au coeur des activités de la plupart des NPU. La deuxième application, nommée IPSec, est basée sur la première et ajoute des fonctionnalités supplémentaires au niveau de la sécurité des données dont l'authentification et le chiffrement DES.

Les différentes étapes proposées par notre méthodologie seront suivies. Tout d'abord, une plate-forme extrêmement simple sera développée et les applications seront complètement réalisées purement en logiciel. Par la suite, le code sera exécuté et profilé sur la plate-forme de base afin d'obtenir une idée plus claire des fonctions à optimiser. Face à ces premiers résultats, une version simple d'un Xtensa avec les options architecturales appropriées sera générée. La troisième étape, sur laquelle nous allons nous attarder plus en détail, se penchera sur la création d'instructions spécialisées permettant d'optimiser les boucles critiques. Finalement, la pertinence d'ajouter des coprocesseurs matériels au lieu d'instructions spécialisées sera étudiée.

3.1 Plateforme initiale

Afin de débuter l'exploration architecturale, une première plate-forme de base doit être développée. La plate-forme la plus simple possible d'un NPU contient un processeur, une mémoire et une interface pour les entrées et les sorties. La figure 3.1 présente cette plate-forme servant de point de départ. Le réseau d'interconnexions est modélisé par un canal SOCP au niveau UTF. Il n'implique donc aucun délai de traitement. Puisque nous ne désirons pas étudier l'impact du NoC pour l'instant, il ne sera pas raffiné lors de l'exploration architecturale. La mémoire, quant à elle, est modélisée comme étant capable d'effectuer une lecture ou une écriture par cycle d'horloge. Comme c'est le cas avec le NoC, il est très facile de modifier la valeur de la latence de la mémoire si jamais nous désirons étudier l'impact que cela pourrait avoir. Les modules d'entrées/sorties SPD (*Simple Packet Device*) permettent quant à eux de recevoir et d'envoyer des paquets à l'extérieur de la plate-forme.

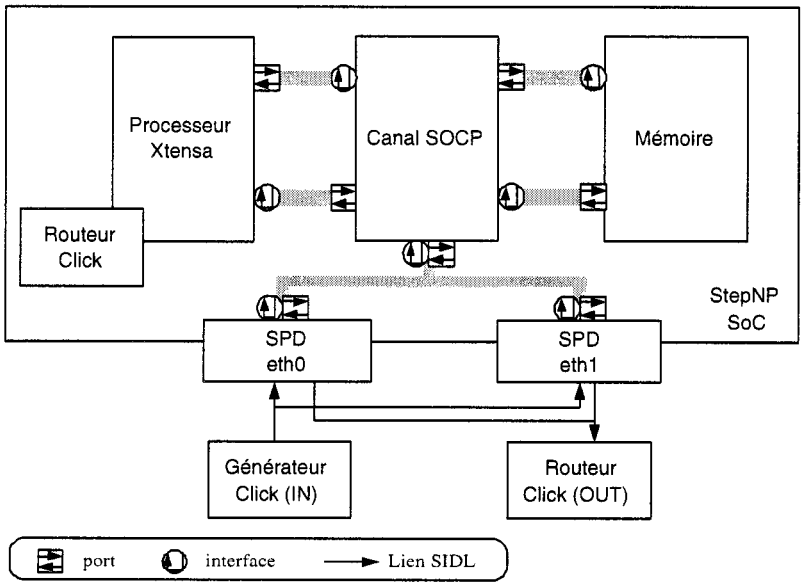


FIGURE 3.1 Plate-forme initiale pour l'exploration architecturale

À la base, aucun coprocesseur n'est utilisé sur la plate-forme. Avec un processeur à usage général, tel un ARM, il serait probablement nécessaire d'utiliser du matériel

spécialisé pour l'épauler et pour atteindre des performances acceptables. Puisque dans ce cas-ci le processeur configurable Xtensa est utilisé, l'emphase sera mise sur l'utilisation d'instructions personnalisées et non sur la création de coprocesseurs. Puisque nous désirons obtenir des premières mesures de performance sans instructions spécialisées, la plate-forme initiale contient un Xtense avec aucun ajout ou extension.

3.1.1 Utiliser Click sur un Xtensa

À la base, Click est conçu pour s'exécuter sur un ordinateur personnel avec Linux comme système d'exploitation. Dans cet environnement, il est possible d'utiliser Click comme une application simple ou encore comme un pilote du système d'exploitation qui peut accéder directement aux cartes réseaux de la machine. Dans notre cas, nous désirons utiliser Click sur un Xtensa sans aucun système d'exploitation. Afin de pouvoir compiler Click dans ces conditions, il est nécessaire d'exclure bon nombre d'éléments qui ont un lien direct avec Linux et de s'assurer que les autres éléments de base peuvent compiler sur le Xtensa. Click a déjà été modifié pour rouler sur un processeur ARM. Puisque le Xtensa, dans sa version de base, est un processeur RISC qui est somme toute comparable au ARM, le travail de conversion pour apporter Click sur un Xtensa est sensiblement le même.

Lorsqu'un logiciel complexe comme Click doit être exécuté sur un nouveau type de processeur, le premier défi est de s'assurer que le compilateur C++ du nouveau processeur supporte toutes les constructions utilisées dans Click. Le second défi consiste à fournir certaines fonctions standard du système d'exploitation Unix qui sont requises par Click et, bien entendu, qui ne sont pas présentes sur le Xtensa sans système d'exploitation. Un exemple de ces fonctions est `ntoh()`, qui permet une conversion de la représentation des mots sur le réseau (*big-endian*) à celle utilisée par l'hôte si nécessaire. La fonction `gettimeofday()`, utilisée par Click pour obtenir des mar-

queurs temporels, est un autre exemple. Il s'agit là d'un travail assez fastidieux et inintéressant à accomplir. Une description plus complète du processus souffrirait immanquablement des mêmes qualificatifs et ne sera donc pas réalisée dans ce mémoire.

Une fois ce travail effectué, différentes configurations Click peuvent être réalisées pour s'assurer du bon fonctionnement du logiciel dans son nouvel environnement. Par exemple, avec le fichier de configuration présenté à la figure 3.2, le Xtensa reçoit un paquet du module d'entrée SPD (décrit à la section 3.1.2) situé à l'adresse `0x6fff0000`. Par la suite l'en-tête Ethernet est supprimée, le contenu du paquet est imprimé puis une nouvelle en-tête est ajoutée et le paquet est envoyé à la sortie.

```
SpdSource(ADDR 0x6fff0000)
    -> Strip(14)
    -> CheckIPHeader(18.26.4.255 2.255.255.255 1.255.255.255)
    -> Print(xtensaReceived)
    -> EtherEncap(0x0800, 00:00:c0:ae:67:ef, 00:00:c0:4f:71:ef)
    -> SpdSink(ADDR 0x6fff0000)
```

FIGURE 3.2 Exemple d'une configuration Click

3.1.2 Modules d'entrée et de sortie

Afin de pouvoir effectuer des simulations représentatives, il est nécessaire de se doter d'un mécanisme pour recevoir des paquets sur la plate-forme et pour les envoyer une fois le traitement terminé. Pour ce faire, l'outil SIDL offert par StepNP est mis à contribution. Un module SystemC nommé SPD (Simple Packet Device) est utilisé dans la plate-forme. Ce module possède une interface SIDL permettant de recevoir et d'envoyer des paquets avec une application externe à la plate-forme simulée. Il est donc possible de concevoir un générateur de paquets qui s'exécute sur la machine hôte de la simulation et qui communique avec le SPD au travers de l'interface SIDL. Un grand nombre de possibilités sont envisageables pour implémenter ce générateur de paquets. Il est possible de se créer soi même un programme qui génère des paquets

semi aléatoires ou encore provenant d'une table de paquets typiques. Il est aussi possible de se connecter directement sur la carte Ethernet de la machine hôte. Une troisième option, que nous avons employée, est de se servir de Click afin de générer les paquets. Pour que cette approche fonctionne, il suffit d'ajouter un élément possédant l'interface SIDL appropriée à l'application Click qui s'exécute sur la machine hôte.

Du côté de la plate-forme, le module SPD est un esclave sur le canal de communication qui est accédé par le processeur pour lire ou écrire un paquet. Ce module possède trois champs qui sont accessibles à partir de trois adresses différentes. Le premier champ est une mémoire tampon capable de contenir un paquet en entier, le second champ (nommé `inReady`) indique si un paquet est présent et prêt à être traité (valeur égale à '1') ou si un nouveau paquet peut être lu du générateur de paquets (valeur égale à '0'). Le dernier champ indique la taille du paquet dans la mémoire tampon. Le processeur lit le champ `inReady` jusqu'à ce qu'il reçoive une réponse positive (un '1'), par la suite il lit la taille du paquet et vide la mémoire tampon. Une fois le paquet reçu, le processeur indique au SPD que le paquet est lu en écrivant '0' dans `inReady`. Aussitôt que `inReady` vaut '0' le SPD reçoit un nouveau paquet du générateur externe, puisque ce dernier espionne aussi le champ `inReady` afin de savoir quand un nouveau paquet peut être généré. Ce mécanisme de scrutation simple assure qu'il y a toujours un paquet disponible pour le processeur et que le débit de traitement maximal du processeur est atteint. La réception des paquets traités par le module de vérification externe, lui aussi implémenté avec Click, s'effectue de manière similaire.

3.2 Spécification et profilage des deux applications réseaux

Maintenant que Click fonctionne sur le Xtensa et que les mécanismes d'entrée/sortie sont bien définis, il est facile de construire des applications plus complexes et de les profiler. Habituellement, l'exploration architecturale repose sur un certain nombre

d'objectifs à atteindre et ces objectifs dictent les métriques que l'on cherche à mesurer lors du profilage [26]. La liste suivante donne quelques uns des objectifs les plus courants.

- **Le coût**. Il est possible d'estimer le coût total en sommant l'investissement nécessaire pour créer ou acheter chacune des parties du SoC ainsi qu'en tenant compte des coûts reliés à la fabrication.
- **La consommation de puissance**. Bon nombre de systèmes embarqués sont portatifs et fonctionnent sur des piles. La réduction de la consommation peut alors devenir l'objectif principal.
- **La performance**. Dans un système traitant beaucoup de données, le débit maximal atteignable peut devenir la préoccupation principale. Dans d'autres cas c'est la latence et le temps de réponse à certains événements qui deviennent l'objectif.
- **La flexibilité**. Tel qu'expliqué dans l'introduction, avoir un bon niveau de flexibilité dans un SoC est un avantage incontestable. Il est cependant difficile de mesurer de manière quantitative un niveau de flexibilité. Souvent, c'est uniquement lorsque vient le temps d'implémenter une fonctionnalité non prévue lors du design initial que l'on découvre le niveau de flexibilité d'un SoC. Néanmoins il est possible de se faire une idée qualitative de la flexibilité entre autre en regardant la facilité avec laquelle le logiciel peut être adapté sur la plate-forme et quelle sont les restrictions au niveau du développement du logiciel.

Dans le cadre de cette recherche nous allons surtout nous préoccuper de la vitesse d'exécution. La puissance n'est pas un très grand problème pour un processeur réseau et le coût n'a pas été pris en compte dans cette étude. Quant à la flexibilité, il est déjà possible d'affirmer que l'utilisation de Click offre un bon niveau de flexibilité. De plus, même avec des instructions spécialisées, le Xtensa garde son noyau d'instructions générales et peut donc encore s'acquitter d'un grand nombre de tâches imprévues.

3.2.1 Application IPv4

La première étape de la méthodologie proposée à la section 2.4 consiste à réaliser toute l'application en logiciel. Cette sous-section présente donc la première application réalisée ainsi que les résultats du profilage initial.

3.2.1.1 Description de l'application

Le protocole IPv4 est le protocole de communication de base sur Internet qui permet le routage des paquets au travers du réseau. La tâche principale à accomplir pour le routeur IPv4 est de trouver quelle est la prochaine destination du paquet. Si le paquet doit passer par un autre routeur il est simplement transféré après avoir subi quelques transformations et si la destination est directement reliée au routeur, le protocole ARP (*Address Resolution Protocol*) est utilisé afin d'obtenir l'adresse réelle du destinataire (adresse Ethernet) à partir de son adresse logique (adresse IP). L'application réalisée permet le transfert d'informations entre deux réseaux locaux (nommés ETH0 et ETH1 sur la figure 3.1) où l'un d'entre eux est relié à l'Internet. L'application est réalisée en connectant différents éléments Click ensemble et elle se base sur une version originalement disponible avec Click sur laquelle nous avons ajouté les éléments d'entrée/sortie SPDsource et SPDsink. Plus de détails sur cette application et sur les différents protocoles impliqués sont disponibles dans [68].

3.2.1.2 Résultats du profilage

Puisque nous désirons principalement optimiser la vitesse d'exécution, la métrique choisie est le nombre de cycles consommés par le processeur pour traiter un paquet. Bien entendu, les performances ne dépendent pas uniquement du nombre de cycles, mais aussi de la fréquence du processeur. La fréquence d'exécution sera prise en

compte à la section 3.4.3. Afin d'obtenir une idée d'où se situe les fonctions critiques, il est utile de déterminer non seulement le nombre de cycles total, mais aussi le nombre de cycles dans chaque fonction. Ce type de profilage est relativement aisé à réaliser. Il suffit, en effet, de noter à chaque cycle de l'ISS l'adresse de l'instruction exécutée. Cette adresse est obtenue en lisant le registre PC du processeur. Par la suite, en comparant ces adresses avec la table symbolique des fonctions dans le fichier binaire exécutable, il est possible de savoir combien de cycles ont été passés dans chacune des fonctions. Le logiciel GNU Profiler (gprof) automatise cette tâche et fournit aussi une multitude d'autres informations en autant que l'on génère un fichier d'information compatible lors de la simulation. L'ISS du Xtensa supporte par défaut le format de fichier de gprof ce qui nous permet de très facilement obtenir des résultats de profilage précis après la simulation. Bien entendu, une application complexe comme celle simulée ici possède des centaines de fonctions et il est donc utile de regrouper les différentes fonctions en catégories afin d'avoir une meilleure vue d'ensemble de la situation. Dans notre cas, sept différentes catégories ont été créées.

1. **Window Overflow et Window Underflow**. Tel qu'expliqué à la section 1.1.1, le Xtensa utilise un mécanisme de fenêtres coulissantes pour les appels de fonctions. Lorsqu'un trop grand nombre d'appels imbriqués se produisent, il est alors nécessaire de sauvegarder les registres d'états et de données dans une pile.
2. **Fonction Push**. Tous les éléments Click implémentent la fonction virtuelle *push* ce qui leur permet d'échanger les paquets peu importe l'ordre dans lequel ils sont connectés les uns aux autres
3. **Gestion de la mémoire**. Cette catégorie regroupe toutes les fonctions qui s'occupent de la gestion dynamique de la mémoire. Ces fonctions sont principalement utilisées pour lire et écrire les paquets dans la mémoire du processeur.
4. **Paquets IN/OUT**. Cette catégorie regroupe les fonctions utilisées pour échanger les paquets avec les interfaces d'entrée/sortie.

5. **Vérification de l'en-tête IP**. Toutes les fonctions de vérification et de mise à jour des différents champs de l'en-tête IP, mis à part le checksum, sont incluses dans cette catégorie.
6. **Checksum**. Cette catégorie inclut le calcul du nouveau checksum après modification de l'en-tête IP.
7. **Autres**. Toutes les autres fonctions sont regroupées ici, cela inclut les recherches dans la table de routage et le traitement des en-têtes Ethernet.

Deux différents types de trafic ont été envoyés à la plate-forme. Le premier est uniquement constitué de paquets de taille minimale (64 octets) et le second comprend des paquets de taille variable (de 64 à 1518 octets), ce qui est plus représentatif d'un trafic réel. Puisque la majorité du traitement s'effectue sur les en-têtes des paquets (qui sont de taille fixe) et non sur les données, les différences observées entre les deux types de trafic sur les temps de calculs devraient être minimales. Cependant le temps nécessaire pour copier les paquets de grande taille en mémoire sera immanquablement plus grand avec des paquets de taille variable. Le tableau 3.1 donne les résultats du profilage initial.

TABLEAU 3.1 Résultats initiaux du profilage de l'application IPv4

Catégories	Temps d'exécution			
	Taille minimale		Taille variable	
	Cycles	Pourcentage	Cycles	Pourcentage
Window Overflow et Underflow	823	22.8	892	14.6
Fonction *Push*	609	19.9	636	10.4
Gestion de la mémoire	898	24.9	3557	58.2
Paquets IN / OUT	173	4.8	190	3.1
Vérification de l'en-tête IP	123	3.4	73	1.2
Calcul du checksum	130	3.6	122	2.0
Autres	852	23.6	642	10.5
Total	3608	100.0	6112	100.0

La première constatation face à ces résultats est que l'utilisation d'une application flexible comme Click a un coût non négligeable. En effet, les débordements de la

fenêtre coulissante sont directement reliés aux multiples appels de fonctions imbriqués de Click, alors que la fonction *push* est, elle aussi, reliée à la structure flexible de Click. L'impact de la taille des paquets sur la quantité de gestion de la mémoire à effectuer est aussi bien illustré par ces résultats. Avec des paquets de taille variable, la gestion de la mémoire occupe prêt de 60% du temps du processeur et les autres tâches sont relégués à l'arrière plan. Dans les prochaines sections, nous allons nous baser sur ces premiers résultats afin de déterminer quelles optimisations devraient être effectuées.

3.2.2 Application IPSec

L'objectif visé avec cette seconde application est de se doter d'un exemple concret qui demande plus de calculs de la part du processeur que l'algorithme de routage simple de IPv4. Le cryptage des données est de plus en plus utilisé sur Internet et demande justement beaucoup de calculs spécialisés. Cette sous-section présente donc une application réseau qui supporte le chiffrement des données.

3.2.2.1 Description de l'application

IPsec (*IP Security*) est un protocole qui ajoute certaines fonctionnalités à IPv4 afin de rendre le transfert des données plus sécuritaire à travers le réseau. Les deux principaux ajouts de IPsec sont l'authentification de l'émetteur et du récepteur ainsi que le chiffrement des données. Cependant, l'application Click que nous avons créée s'occupe principalement du chiffrement de type DES (*Data Encryption Standard*). Les paquets subissent le même traitement que dans l'application IPv4 à une exception près : avant d'envoyer le paquet à la sortie, l'algorithme DES est appliqué, puis le paquet est encapsulé avec une nouvelle en-tête. L'ajout d'une nouvelle en-tête, qui est géré par le protocole *Encapsulating Security Payload* (ESP), permet de protéger

tout le paquet et non seulement les données. Notons finalement que, typiquement, c'est plutôt l'algorithme triple-DES, ou 3DES, qui est appliqué. Il s'agit là d'une simple modification qui consiste à appliquer trois fois DES sur les données afin de rendre plus difficile le décryptage par une personne non autorisée.

DES fonctionne avec une clé publique symétrique, c'est-à-dire que l'émetteur et le récepteur possèdent tous les deux la même clé de 64 bits pour le cryptage et le décryptage des données. Cette clé est générée au début de la communication et possède une durée de vie donnée. Le processus d'authentification s'occupe de gérer l'échange initial de la clé. Pour simplifier l'application, cette partie de IPsec n'a cependant pas été implémentée et notre application utilise une clé unique pour le chiffrement de tous les paquets. Puisque l'authentification s'effectue uniquement au début d'une séquence de transactions, cette omission ne change pas les résultats relatifs au transfert des données en tant que tel. Les données sont traitées par bloc de 64 bits et chaque bloc subit des permutations et des manipulations complexes qui dépendent de la clé utilisée. Afin d'augmenter la sécurité, chaque bloc de 64 bits n'est pas traité indépendamment, ils sont plutôt chaînés les uns aux autres avec un mode d'opération nommé CBC (*Cypher Block Chaining*). Cet ajout fort simple consiste à insérer au début du transfert 64 bits aléatoires, puis à réaliser entre les blocs cryptés voisins un « ou exclusif ». Plus de détails sur le fonctionnement de l'algorithme DES sont donnés dans l'annexe V

3.2.2.2 Résultats du profilage

Puisque l'algorithme DES requiert plusieurs manipulations au niveau des bits qui sont difficiles à réaliser avec un processeur conventionnel, il est attendu que cette portion de l'application va demander beaucoup de temps de calcul. C'est pourquoi une huitième catégorie est ajoutée pour les résultats du profilage : le chiffrement DES-CBC. Les temps d'exécution obtenus, encore une fois avec des paquets de taille

minimale et des paquets de taille variable, sont données dans le tableau 3.2. Comparativement à l'application IPv4, l'augmentation du nombre de cycles requis pour la gestion de la mémoire s'explique par le fait que la nouvelle fonction de chiffrement génère plus de données à transférer (à cause des en-têtes ajoutées). Quant aux fonctions *push* et à la gestion des débordements de la fenêtre coulissante, l'augmentation du nombre de cycles s'explique par l'ajout d'éléments Click supplémentaires pour effectuer le cryptage. Cependant, en terme de pourcentage, le chiffrement des données occupe maintenant une énorme partie du temps de calcul du processeur, jusqu'à 86.8% avec des paquets de taille variable.

TABLEAU 3.2 Résultats initiaux du profilage de l'application IPSec

Catégories	Temps d'exécution			
	Taille minimale		Taille variable	
	Cycles	Pourcentage	Cycles	Pourcentage
Chiffrement DES-CBC	6177	48.9	68183	86.8
Window Overflow et Underflow	1443	11.7	1492	1.9
Fonction *Push*	1019	8.3	1131	1.4
Gestion de la mémoire	1769	14.3	5514	7.0
Paquets IN / OUT	201	1.6	204	0.3
Vérification de l'en-tête IP	94	0.8	94	0.1
Calcul du checksum	184	1.5	189	0.2
Autres	1457	12.9	1744	2.3
Total	12344	100.0	78551	100.0

3.3 Premières optimisations

Avant de se lancer dans la création d'instructions spécialisées, ou même de coprocesseurs, il importe, tel que proposé par la méthodologie présentée à la section 2.4, de se pencher sur des optimisations au niveau du logiciel et aussi sur les options configurables du processeur. Nous procédons avec ces étapes en premier puisqu'elles peuvent permettre d'obtenir des gains de performance tout en nécessitant peu d'effort.

3.3.1 Optimisations logicielles

Des modifications trop ambitieuses de l'application originale peuvent être très longues à réaliser et ne sont pas appropriés lors de la phase d'exploration architecturale et de partitionnement. Il est en effet peu utile de se casser la tête à réécrire une fonction qui finira peut-être par être implémentée avec du matériel, soit sous la forme d'une instruction spécialisée, soit sous la forme d'un coprocesseur. Une première modification triviale que l'on peut apporter à Click est de compiler uniquement les éléments requis pour l'application. Cela réduit la phase d'initialisation du programme et peut aussi diminuer le taux d'échec en mémoire cache puisque la taille de l'exécutable est plus petite.

Une deuxième option est d'éliminer les appels de fonctions virtuelles de Click. Tous les éléments Click héritent de la même classe C++ de base qui définit l'interface des éléments, ce qui inclut la fonction *push*. Lorsqu'un élément doit passer le paquet à l'élément suivant dans la configuration, il appelle la fonction *push* de la classe de base puisqu'il ne connaît pas le type de la classe enfant. Cette manière de procéder fait appel au polymorphisme du C++ et utilise des tables de fonctions virtuelles afin de déterminer quelle fonction doit réellement être appelée au moment de l'exécution. Puisque la séquence d'appels de fonction est uniquement déterminée à l'exécution, et non à la compilation, une perte de performance est encourue. Les concepteurs de Click, conscients de ce problème, offrent un outil qui permet de « dévirtualiser » une configuration donnée. Cet outil effectue une analyse syntaxique du code et remplace toutes les fonctions virtuelles par des appels de fonctions déjà résolus. Tel que présentée à l'annexe VI la dévirtualisation permet un gain de 21% pour la catégorie « fonction *push* » dans l'application IPv4.

Une autre approche, utilisée par les concepteurs de StepNP, pour optimiser Click est d'éliminer entièrement la phase d'initialisation de Click, qui est très longue. Pour ce faire, le fichier de configuration listant les éléments utilisés et leurs connexions n'est

plus utilisé. La configuration est plutôt réalisée à la main dans un fichier principal C++ où les différents objets (correspondant à un élément) sont créés et connectés ensemble. Cela évite d'avoir à analyser un fichier de configuration en cours d'exécution et permet au compilateur C++ d'effectuer beaucoup plus d'optimisations, puisqu'un grand nombre d'informations (incluant les appels à la fonction *push*) sont alors connues au moment de la compilation. Malheureusement, par manque de temps cette modification à Click, nommée NanoClick, n'a pas été testée dans le cadre de ce mémoire.

3.3.2 Configuration du processeur

Choisir les bonnes options lors de la configuration du processeur est une étape simple qui peut engendrer des gains de performance. Il est donc intéressant de se pencher sur ces options dès le début de l'exploration architecturale. Les différents paramètres configurables du Xtensa ont été donnés au tableau 1.1 et les différents blocs logiques qui peuvent être utilisés ou non sont listés à la section 1.1.1.2.

Une première option évidente à choisir est d'utiliser le mode d'opération gros-boutiste (*big-endian*) pour l'encodage des données. Puisque les données transigées sur un réseau IP sont toutes encodées en mode gros-boutiste, un processeur fonctionnant en mode petit-boutiste (*little-endian*) doit effectuer une conversion sur tous les mots de 32 bits reçus et envoyés afin d'inverser l'ordre des octets. Il est possible de faire une instruction spécialisée qui effectue cette conversion en un cycle d'horloge ou encore de créer un module de conversion matériel qui serait branché sur le bus du processeur. Simplement choisir l'option gros-boutiste reste la meilleure approche puisqu'elle n'implique aucun coût additionnel.

Une autre option qui offre des gains intéressants avec nos applications est l'augmentation du nombre de registres physiques de 32 à 64. Cela permet d'avoir plus

d'espace pour la fenêtre coulissante et réduit le nombre de débordements (Window Overflow et Window Underflow). Il en résulte un gain de 78% pour cette catégorie avec l'application IPv4.

La taille des mémoires caches ainsi que le type d'accès aux mémoires caches doit aussi être sélectionné à cette étape. Une cache de grande taille offre généralement des meilleures performances mais coûte plus cher et demande plus de puissance qu'une plus petite cache. De plus, arrivé à une certaine taille, une augmentation de la cache offre des gains négligeables. Il importe donc de faire plusieurs simulations avec l'application cible afin de trouver le compromis idéal. Pour ce travail, nous nous sommes cependant contentés de sélectionner une cache suffisamment grande pour limiter au maximum le nombre d'échecs (moins de 1%).

Le tableau 3.3 résume les caractéristiques du processeur Xtensa généré. Notons qu'aucune unité fonctionnelle tels qu'un multiplicateur ou une unité point flottant n'a été ajoutée puisque les deux applications n'effectuent pas de calculs mathématiques de ce type. Le Xtensa offre aussi plusieurs options au niveau de la gestion de la mémoire, par exemple en proposant l'utilisation d'une unité de gestion de la mémoire pour les adresses virtuelles. Cependant, aucune de ces options ne permet d'améliorer le problème de gestion de la mémoire que nous avons rencontré dans les deux applications.

TABLEAU 3.3 Caractéristiques du processeur Xtensa utilisé

Paramètre	Valeur
Cache d'instruction	8 Ko, Associative par deux
Cache de données	8 Ko, Associative par deux
Taille de la fenêtre coulissante	64 registres
Taille du bus externe	32 bits
Nombre d'interruptions	Aucune
Nombre de compteurs	Aucun
Unités fonctionnelles spécialisées	Aucune
Ordonancement de la mémoire	Big-endian

3.4 Création d'instructions spécialisées

Une fois le processeur correctement configuré, la troisième étape de notre méthodologie consiste à créer les instructions spécialisées appropriées. Pour ce faire, certaines fonctions critiques ont d'abord été identifiées et, par la suite, les étapes suivantes ont été suivies :

1. La fonction est extraite de Click afin d'être simulée et profilée indépendamment.
2. Une ou plusieurs instructions TIE sont créees pour remplacer certaines parties de la fonction.
3. La nouvelle fonction optimisée et l'ancienne sont simulées afin de mesurer les gains et s'assurer que les deux donnent les mêmes résultats, cela permet de valider les instructions TIE.
4. La nouvelle fonction optimisée est introduite dans Click et l'application entière est simulée à nouveau.

De plus, lors de la création d'une nouvelle instruction, certains facteurs doivent être pris en considération. Tout d'abord, le langage TIE n'impose aucune restriction sur la complexité des instructions réalisées. Il est donc possible de créer une instruction qui va générer des unités fonctionnelles très lentes et qui vont ralentir la fréquence d'opération du pipeline à des niveaux inacceptables. Pour contourner ce problème, il est possible de créer une instruction multi-cycles ou encore d'implémenter la fonctionnalité avec plusieurs instructions différentes. L'autre point à considérer est qu'à chaque fois que l'on effectue une opération dans une instruction TIE (une addition par exemple) un nouveau module matériel est ajouté par défaut pour réaliser cette opération (un additionneur dans ce cas-ci). Il est possible de partager des blocs de logique déjà existants avec une nouvelle instruction si l'on respecte certaines constructions spécifiques du langage TIE. Cela permet de réduire la quantité de logique insérée dans le processeur.

3.4.1 Pour l'application IPv4

Lorsque l'on analyse les résultats du profilage, la catégorie qui consomme le plus de cycles de simulation est la gestion de la mémoire. Cependant, comme il sera expliqué à la section 3.5 les instructions TIE ne sont pas adaptées pour résoudre ce problème. Les catégories « fonction *push* » et « Window Overflow / Underflow » ont déjà été optimisées respectivement au niveau du logiciel et des options de configuration du processeur.

La catégorie qui semble la plus se prêter aux instructions TIE est le calcul du checksum, aussi appelé somme de contrôle. Il s'agit d'un champ de 16 bits dans l'en-tête IP qui permet de vérifier si l'en-tête est corrompue. Pour chaque paquet reçu, le checksum est donc calculé et validé. De plus, avant d'envoyer le paquet à la sortie un nouveau checksum doit être recalculé si jamais le routeur a modifié un des champs de l'entête (le champ TTL, Time To Live, par exemple). L'algorithme du checksum consiste simplement à diviser l'en-tête IP en mots de 16 bits et puis à les additionner en complément à 1. Pour réaliser le checksum sur un processeur fonctionnant en complément à 2, il faut conserver les retenues et les additionner avec la somme provisoire obtenue. L'algorithme utilisé par Click est donné à la figure 3.3. Il utilise un accumulateur de 32 bits pour conserver la somme de tous les mots de 16 bits. À la fin de l'addition, les 16 bits les plus significatifs de l'addition, qui correspondent à la retenue, sont additionnés avec les 16 bits les moins significatifs (qui contiennent la somme en complément a 2). Notons finalement que puisque le champ checksum fait lui-même partie de l'en-tête, la norme RFC1071 [11] spécifie qu'il doit être fixé à 0 avant d'effectuer le calcul pour la première fois.

La vaste majorité du temps, la taille de l'en-tête IP est de 20 octets (5 mots de 32 bits), il serait avantageux de calculer le checksum en 5 additions sur des mots de 32 bits, mais cela est impossible puisque les retenues sont alors perdues. L'instruction TIE développée contourne cependant le problème puisqu'elle reçoit 32 bits de l'en-

```
// count contient le nombre d'octets de l'en-tête
// addr est un pointeur sur l'en-tête
uint32_t sum = 0;
while( count > 1 ) {
    sum += * (unsigned short) addr++;
    count -= 2;
}

// Addition de la somme et de la retenue
sum = (sum & 0xffff) + (sum >> 16);
sum += (sum >> 16);

// Tronquer la réponse à 16 bits et l'inverser
uint16_t answer = ~sum;
return answer;
```

FIGURE 3.3 Code utilisé par Click pour calculer le checksum

tête par appel. Le mot est divisé en deux parties de 16 bits qui sont additionnées ensemble pour obtenir un résultat sur 18 bits. Par la suite, les deux bits de la retenue sont ajoutés au résultat. Cette instruction, présentée à la figure 3.4, implique donc deux additionneurs en cascade. Il aurait été possible de réaliser une instruction TIE qui reçoit l'en-tête par mots de 64 bits puisqu'une instruction assembleur du Xtensa peut recevoir deux registres d'entrées comme argument. Il aurait aussi été possible d'ajouter un registre spécialisé de 20 octets au processeur et, une fois rempli avec l'en-tête, de calculer le checksum en un seul cycle. Mais ce faisant, nous serions tombés dans le piège décrit précédemment, c'est-à-dire créer une instruction qui ralentit le pipeline à cause de son trop grand nombre d'opérations séquentielles. Un checksum travaillant sur 20 octects en un seul cycle a été réalisé et selon l'estimation de l'outil de synthèse, son chemin critique était de 8 ns.

Une fois l'instruction TIE complétée, le compilateur TIE de Tensilica génère automatiquement une macro en langage C qui permet d'accéder à cette instruction à partir du code C/C++ sans avoir à passer par des sous-routines en assembleur. La figure 3.5 montre la nouvelle fonction qui utilise cette instruction TIE. L'addition de la retenue est maintenant gérée au fur et à mesure, et non à la fin de l'algorithme. De plus, la boucle principale s'exécute deux fois moins souvent puisque le paquet est

```
// Définition de l'encodage de l'instruction et des entrées/sorties
opcode IPCKSUM1 op2=4'b0000 CUST0
iclass ip_check_sum {IPCKSUM1} {out arr, in ars, in art}

reference IPCKSUM1 {
    wire [17:0] sum;
    wire [15:0] final;
    assign sum = ars[31:16] + ars[15:0] + art[15:0];
    assign final = sum[15:0] + sum [17:16];  // Add. de la retenue
    assign arr = {16'h0000,final};
}
```

FIGURE 3.4 Instruction TIE servant a calculer le checksum

traité par bloc de 32 bits.

```
uint32_t sum = 0;
while (nleft > 1)  {
    sum = IPCKSUM1(*addr++,acc);
    nleft -= 4;
}
return ~sum;
```

FIGURE 3.5 Nouvelle fonction Click qui utilise l'instruction TIE

Cette fonction optimisée peut être utilisée pour calculer le checksum sur un nouveau paquet ou encore lorsque l'adresse de destination est modifiée. Le cas le plus fréquent qui entraîne une modification de l'en-tête et requiert un nouveau calcul du checksum est cependant la décrémentation du champ TTL. Le champ TTL indique le nombre de sauts que le paquet peut faire dans le réseau et chaque routeur qu'il traverse doit soustraire 1 avant d'envoyer le paquet à la sortie. Dans ce cas, il est possible de calculer le nouveau checksum directement à partir de l'ancien sans avoir besoin de lire toute l'en-tête. Une instruction TIE a donc été ajouté pour réaliser cet algorithme, décrit dans le RFC1624 [70], en 1 cycle d'horloge. La figure 3.6 donne le code TIE qui implémente cette fonction et s'occupe aussi, en parallèle, de soustraire 1 au TTL qui est sauvegardé dans un registre dédié.

Les deux instructions TIE pour le calcul du checkum offrent des gains intéressants.

```
opcode CKSUMUP op2=4'b0010 CUST0
iclass ip_check_sum1 {CKSUMUP} {out arr, in ars} {inout IPTTL}

state IPTTL 8                // Nouveau registre contenant le TTL
user_register TTL 0 {IPTTL} // Nom symbolique accessible en C/C++

reference CKSUMUP {
  wire [15:0] tmp = ~ars[15:0];
  wire [16:0] sum = tmp + 16'hFEFF;
  assign arr=(IPTTL > 1) ? {16'h0000, ~(sum[15:0]+sum[16])} :32'd0;
  assign IPTTL = (IPTTL > 1) ? IPTTL - 1 : 8'd0;
}
```

FIGURE 3.6 Instruction TIE servant mettre à jour le checksum

Pour le calcul du checksum, le nombre de cycles requis passe de 142 à 28, soit un gain de 5. Alors que pour la mise à jour du checksum, le nombre de cycles requis passe de 70 à 36, soit un gain de 2.

Malheureusement, l'application IPv4 n'offre pas beaucoup d'autres fonctions pour lesquelles des instructions TIE serait nettement avantageuses. Cela est dû au fait que le protocole IPv4 est constitué d'une multitude de petites tâches simples à réaliser qui se prêtent mal à une implémentation spécialisée ou encore qui ne sont pas appelées fréquemment. Notons cependant que la tâche la plus complexe de l'application IPv4, la recherche dans une table de routage pour associer une adresse à une destination, a été simplifiée dans notre exemple. La table utilisée ne comprenait que quelques entrées (comparativement à des milliers dans un routeur typique) et demandait donc peu de temps de calcul. Lorsque des tables complexes doivent être utilisées, il existe déjà un grand nombre de coprocesseurs pouvant s'acquitter de cette tâche.

3.4.2 Pour l'application IPSec

Alors que l'application IPv4 offrait peu de cas où il est avantageux d'utiliser des instructions spécialisées, il en va tout autrement pour l'application IPSec et l'algorithme de chiffrement DES qu'elle implémente. Notons dans un premier temps que

les optimisations logicielles et les choix faits pour la configuration du logiciel, décrits respectivement aux sous-sections 3.3.1 et 3.3.2, s'appliquent aussi bien pour IPSec que pour IPv4. De plus, l'algorithme logiciel pour le chiffrement DES implémenté dans Click est déjà efficace et tenter de l'optimiser ne serait pas du temps bien investi. L'étape suivante est donc de regarder les instructions spécialisées.

L'algorithme DES, décrit à l'annexe V est fastidieux à implémenter en logiciel. Il est cependant un candidat idéal pour des instructions TIE principalement parce qu'il utilise uniquement des opérations simples à réaliser en matériel, mais mal adaptées au logiciel. Par exemple, parmi ces opérations se trouvent :

- des permutations où tous les bits du mot (de 32, 48 ou 64 bits) changent de position ;
- des rotations de bits sur des mots de 28 bits ;
- des « ou exclusif » sur des mots de 48 bits ;
- la décomposition d'un mot de 48 bits en 8 mots de 6 bits chacun selon un patron donné.

En matériel, permuter les bits d'un mot peut se faire simplement en inversant l'ordre des fils qui se connectent au registre sauvegardant la nouvelle valeur, il s'agit donc d'une opération à toute fin pratique « gratuite ». En logiciel, un très grand nombre de cycles sont nécessaires pour réaliser cette permutation à l'aide des opérations de base du jeu d'instructions.

Afin d'accélérer le chiffrement, 4 instructions TIE ont été réalisées. La première permet de générer les 16 clés intermédiaires à partir de la clé initialement reçue. Ces 16 clés de 48 bits chacune sont sauvegardées dans des registres internes au processeur. Les trois autres instructions s'occupent du chiffrement en tant que tel en utilisant les clés intermédiaires ainsi que des tables de conversion (look-up table) qui ont aussi été ajoutées dans le processeur. Le code source des instructions TIE ainsi que le code Click modifié pour tirer avantage de ces nouvelles instructions se trouvent à l'annexe V. Cette annexe contient aussi le code utilisé pour valider le bon

fonctionnement des instructions TIE.

Pour la génération des clés, le gain obtenu est très intéressant puisque le nombre de cycles requis est passé d'environ 2300 à 10, pour un gain de 230. Du côté du chiffrement lui-même, le nombre de cycles nécessaires pour traiter un bloc de 64 bits est passé de 900 à 62 pour un gain de 14,5. Si l'on considère la nouvelle fonction Click qui encrypte un paquet de 84 octets au complet avec des clés déjà générées, le gain est de 13,6 (518 cycles au lieu de 7061). Il est aussi intéressant de noter que plus la taille du paquet augmente, plus les gains sont importants.

3.4.3 Résultats de synthèse des instructions

Tel que mentionné précédemment, il est important de s'assurer que les instructions ajoutées au processeur ne ralentissent pas trop sa fréquence d'opération et aussi n'augmente pas la surface ou la puissance consommée par le processeur au delà du seuil acceptable pour le circuit réalisé. Bien que la licence universitaire de Tensilica dont nous disposons ne permet pas de synthétiser l'ensemble du processeur, il est possible de synthétiser uniquement les unités fonctionnelles ajoutées par les instructions TIE. Lorsque l'instruction est compilée, un fichier Verilog qui contient la logique à ajouter dans le processeur est généré et ce fichier peut être synthétisé avec des outils standards. Tensilica fournit aussi des scripts qui permettent de faciliter ce processus. Les résultats des synthèses effectuées avec une technologie 0.18um sont donnés dans le tableau 3.4.

TABLEAU 3.4 Résultats de synthèse des instructions TIE

Instructions TIE	Délais (ns)	Surface (um^2)
Checksum	1.8	110
Checksum update	1.7	12245
DES	2.3	289400
Toutes	2.3	294500

Selon les estimateurs disponibles sur le site Internet de Tensilica, le processeur Xtensa de base utilisé pour le profilage initial de nos deux applications peut être cadencé à 250 MHz (période de 4 ns) et occupe environs 2.2 mm^2 lorsque réalisé dans une technologie 0.18um. La synthèse des instructions TIE n'a pu être réalisée en tenant compte d'un modèle de délais pour les fils (*no wire load model*) et les résultats du tableau 3.4 sont donc optimistes. De plus, le code RTL complet du processeur n'était pas disponible et les nouvelles instructions n'ont pas pu être synthétisées avec l'ensemble du processeur. Il reste néamoins que les résultats obtenus donnent une idée de l'ordre de grandeur auquel nous sommes en droit de s'attendre. Les délais restent cependant bien en deça de 4 ns, ce qui indique que les instructions TIE ne ralentiront pas le processeur en entier. L'instruction checksum demande beaucoup moins de surface que les deux autres puisqu'elle n'ajoute aucun registre supplémentaire dans le processeur, de plus elle réutilise les additionneurs déjà présents dans l'unité arithmétique du processeur.

3.5 Utilisation d'un coprocesseur

La quatrième et dernière étape de la méthodologie, consiste à se pencher sur la pertinence d'utiliser un coprocesseur pour accélérer certaines tâches qui ne se prêtent pas très bien à des instructions spécialisées. Tel que décrit dans les tableaux 3.1 et 3.2, la catégorie « gestion de la mémoire » occupe une part importante du temps du processeur et n'a pas été accélérée lors des étapes précédentes. En analysant l'application il apparaît que deux opérations comptent pour une bonne partie des accès mémoire du processeur.

1. L'alignement des paquets. Le processeur Xtensa ne supporte pas les accès non-alignés, c'est-à-dire que lorsqu'il fait des accès de 4 octets en mémoire, l'adresse doit toujours être un multiple de 4 afin d'être alignée sur une case mémoire. La seule manière de réaliser un accès non-aligné est de lire 2 mots de 4 octets et

ensuite de regrouper les octets désirés dans un seul mot. Avec les applications IPv4 et IPSec, une des premières opérations réalisées par Click est d'enlever l'en-tête Ethernet afin d'accéder au paquet IP. Puisque l'en-tête Ethernet a toujours une taille de 14 octets, le reste du paquet IP n'est plus aligné en mémoire et il devient pénible d'y accéder puisque la lecture d'un mot non aligné demande 2 accès en mémoire. Dans le cas du processeur Xtensa les accès non alignés ne sont tout simplement pas supportés. Pour corriger cela, Click effectue un alignement qui consiste à déplacer le paquet en entier (l'en-tête IP et les données) afin qu'il soit aligné à nouveau.

2. Le transfert avec les interfaces. Lorsqu'un nouveau paquet est disponible, le processeur le copie de l'interface d'entrée vers la mémoire et le processus inverse doit être effectué vers l'interface de sortie lorsque le traitement est terminé.

Nous nous sommes d'abord penchés sur la possibilité d'ajouter une instruction TIE qui s'occupe de faire des accès mémoire non-alignés. Cette tâche n'est cependant pas simple. Dans un premier temps, même avec une instruction TIE, le mécanisme d'accès à la mémoire du processeur ne peut pas être modifié. Il est donc nécessaire de créer une instruction qui effectue deux lectures et qui combine par après les octets désirés dans un mot. Ces deux lectures ne peuvent pas avoir lieu simultanément ce qui oblige à utiliser une instruction multi-cycle. Dans le cas d'une écriture, le processus est relativement similaire à l'exception près qu'il faut faire bien attention d'utiliser les opérations d'écriture de 8 et 16 bits natives du Xtensa afin d'aller placer les 4 octets non-alignés dans les deux cases mémoires de destination sans écraser les données adjacentes. Ces instructions ne seraient pas très rapides et le gain de vitesse ne serait donc pas très intéressant et même si des instructions de chargement et de rangement efficaces pouvaient être créées, il faudrait faire face à un nouveau problème. Des macros C sont fournies pour permettre d'inclure des nouvelles instructions TIE explicitement dans le code C/C++, tel qu'illustré à la figure 3.5. Cependant, les fonctions de gestion de la mémoire utilisées par Click font partie de la bibliothèque standard du langage C et aller les modifier pour qu'elles utilisent les

nouvelles instructions ne serait pas une mince tâche. Le seul cas où un code C/C++ peut automatiquement être compilé (sans être modifié préalablement) pour utiliser des instructions TIE spécialisées est lorsque l'on déclare un nouveau type de registres généraux, par exemple une banque de 16 registres généraux de 64 bits chacun. Il est à ce moment-là possible de définir un nouveau type de donnée C (que l'on nomme par exemple `myInt64`) et les opérations effectuées sur ces registres (telles une lecture, une écriture ou une addition) seront automatiquement implémentées par l'instruction TIE correspondante au moment de la compilation. Dans notre cas, nous effectuons des accès standards de 32 bits et cette approche est difficilement utilisable.

Dans cette situation, il apparaît donc que l'utilisation d'un coprocesseur est plus appropriée. Pour ce faire, une petite mémoire de 4 ko a été connectée sur le port XLMI du processeur. L'interface XLMI est indépendante de l'interface mémoire du processeur. Pour le processeur, un accès sur cette interface n'implique jamais d'accès à la mémoire cache ce qui nous permet de placer un coprocesseur devant la mémoire qui s'occupe de l'alignement. Lorsqu'un accès mémoire non aligné est généré sur le port XLMI, le coprocesseur l'intercepte et s'occupe de retourner les 4 octets désirés. Pour régler la deuxième source d'accès mémoire indésirable de la part du processeur (les transferts avec les interfaces), un deuxième coprocesseur qui s'occupe de remplir la mémoire des paquets a été ajouté. Ce coprocesseur, décrit en SystemC, agit comme un module DMA (*Direct Memory Access*) standard et, sous les ordres du processeur, il s'occupe de transférer un bloc de données consécutives entre les interfaces et la mémoire des paquets. La figure 3.7 montre l'architecture de cette nouvelle plate-forme, alors que [68] offre plus de détails sur le fonctionnement des coprocesseurs. Ces deux coprocesseurs permettent un gain très intéressant de 12.8 pour la catégorie « gestion de la mémoire » lorsque des paquets de tailles variables sont utilisés avec l'application IPv4. Un bon gain (9.8) est aussi observé avec l'application IPSec.

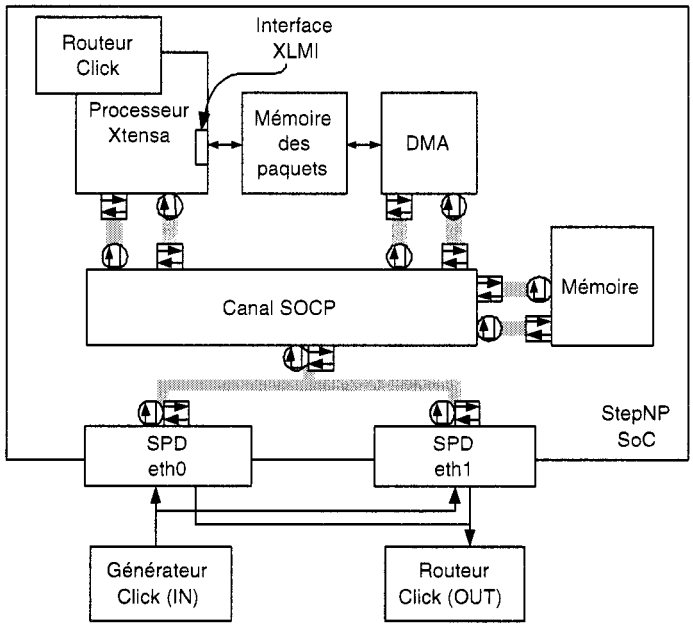
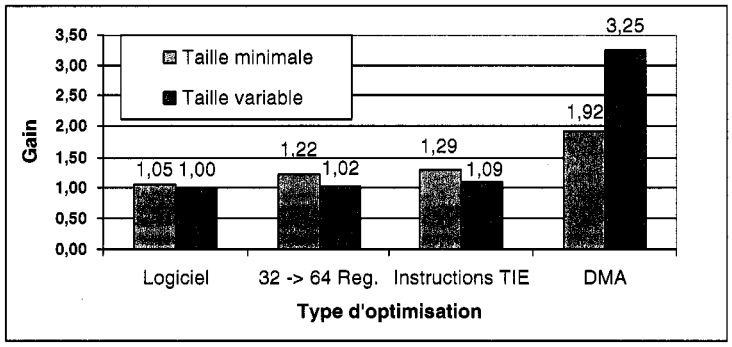


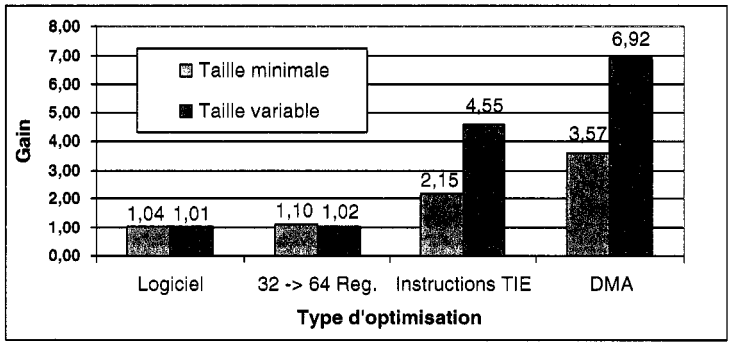
FIGURE 3.7 Architecture de la plate-forme optimisée

3.6 Résultats globaux

Les optimisations réalisées au travers des quatre grandes étapes de notre méthodologie ont offert des gains parfois très appréciables, parfois plus négligeables, pour chacune des 8 catégories d'opérations identifiées (section 3.2.1.2). Le détail de ces gains est présenté dans [69] ainsi que dans l'annexe VI. La figure 3.8 présente quant à elle les gains globaux obtenus en simulant les deux applications après chaque type d'optimisation. Un gain est calculé en faisant le rapport du nombre de cycles requis pour traiter un paquet après chacune des optimisations comparativement au nombre de cycles requis sur la plate-forme initiale sans aucune optimisation. Ces gains sont cumulatifs, c'est-à-dire que pour chaque nouvelle optimisation, les optimisations précédentes sont toujours effectives. Avec cette plate-forme, si l'on suppose que le processeur est cadencé à 400 MHz (technologie 0.13um), il est possible de traiter un débit de 109 Mbps avec des paquets de taille minimale et un débit de 1.16 Gbps avec des paquets de taille variable. Lorsque l'application IPSec est considérée, ces débits chutent à 57.1 Mbps et 193 Mbps.



(a) IPv4



(b) IPSec

FIGURE 3.8 Gains globaux pour les deux applications

Ces résultats montrent clairement que les gains offerts par les instructions TIE sont appréciables dans le cas de l'application DES et moins impressionnants pour l'application IPv4. Étant donnée que l'application IPv4 ne comprend pas un petit nombre de tâches qui consomme la majorité du temps, il aurait fallu créer un très grand nombre d'instructions TIE pour aller chercher de meilleurs gains.

3.6.1 Avantages des processeurs configurables

Le chiffrement DES aurait très bien pu être réalisé par un coprocesseur spécialisé. Avec un tel processeur il serait probablement possible d'extraire encore plus de parallélisme de l'algorithme et d'obtenir un résultat plus rapidement. Cependant l'utilisation d'un processeur configurable a l'avantage de permettre l'ajout de nouvelles

unités fonctionnelles très rapidement ; la création des instructions TIE présentées dans cette section a demandé de 2 à 3 semaines. De plus, la génération du matériel et l'intégration dans le processeur se fait automatiquement. Cela réduit grandement le temps de conception et limite les erreurs possibles.

Un autre avantage des instructions spécialisées est qu'elles s'intègrent facilement dans le code de l'application originale. Avec un coprocesseur, il est nécessaire d'ajouter du code afin de gérer les communications et la synchronisation entre le processeur et le coprocesseur. Cette communication crée aussi un trafic supplémentaire sur le canal de communication ce qui peut entraîner une dégradation des performances. Ce sont ces raisons qui font que lors de la recherche architecturale, la création d'instructions spécialisées devrait être favorisée et l'utilisation d'un coprocesseur peut être envisagé si jamais les instructions spécialisées s'avèrent insuffisantes pour atteindre nos objectifs.

3.6.2 Améliorations possibles

La plate-forme présentée à la figure 3.7 est un bon point de départ, mais plusieurs améliorations sont encore possibles. La plus évidente est d'utiliser plus d'un processeur afin d'augmenter le débit maximal. La vaste majorité des paquets reçus par un routeur sont indépendants les uns des autres. En utilisant plus d'un processeur, il est donc possible de diviser le flot de données et d'exploiter ainsi le parallélisme inhérent au traitement de paquets. Pour ce faire, il est cependant nécessaire d'avoir un réseau d'interconnexions et des modules d'entrée/sortie capables de supporter un plus haut débit. La latence des communications et aussi celle des accès mémoires limitent donc le nombre de processeur que l'on peut ajouter à la plate-forme actuelle. En effet, si l'on ajoute trop de processeurs sur la plate-forme, ils vont passer la majorité de leur temps à attendre après le NoC qui sera alors surchargé. Il serait donc intéressant de réaliser une plate-forme avec un modèle d'interconnexions plus réaliste et d'effectuer

des simulations pour vérifier les performances atteignables avec différents nombre de processeurs.

Lorsque l'on configure le processeur Xtensa, il est possible de configurer la largeur du bus d'accès mémoire (32, 64 ou 128 bits). Il serait intéressant de vérifier si le fait de lire le paquet par blocs de 16 ou 8 octets au lieu de 4 permettrait d'augmenter les performances. Cela pourrait s'avérer particulièrement utile au niveau de l'interface XLMI et de la mémoire des paquets puisque le chiffrement DES travaille déjà sur des blocs de 64 bits.

Finalement, afin de faciliter cette première phase d'exploration architecturale, plusieurs simplifications ont été faites. Par exemple, l'application IPv4 utilise une table de routage réduite ce qui n'est pas très représentatif. Du côté de l'application IPSec toute la phase d'authentification et de gestion des clés a été enlevée. Finalement, le canal de communication a été simulé sans aucune latence. En combinant cela au mode d'opération des modules SPD d'entrée/sortie, nous obtenons un processeur qui est utilisé au maximum. Cela donne une mesure des performances maximales atteignables. Ajouter une latence dans le canal réduirait automatiquement les performances. Ce problème est abordé dans le chapitre 4.

CHAPITRE 4

MODÈLE D'UN PROCESSEUR HMT

Le dernier chapitre de ce mémoire se penche sur une seconde technique pour optimiser un processeur réseau : l'utilisation d'un processeur avec support pour le traitement multiprocessus. Cette technique est différente de celle présentée au chapitre 3 (des instructions spécialisées), mais néanmoins complémentaire puisque les deux approches peuvent être combinées. Tel qu'expliqué à la section 1.2, un processeur HMT aide à masquer un problème de taille dans la conception d'un NPU rapide : la latence des communications. Dans ce chapitre, cette problématique sera d'abord décrite. Par la suite, différentes techniques de modélisation d'un processeur HMT à partir d'un ISS sont données et le cas du Xtensa sera aussi étudié plus en détails. De plus, certains tests simples illustrant l'avantage d'utiliser un processeur HMT sont présentés.

4.1 Avantages d'un processeur HMT

Une des architectures typiquement rencontrée dans un NPU comprend plusieurs processeurs identiques qui exécutent le même code pour le traitement de paquets. Ces processeurs peuvent être des processeurs avec des instructions spécialisées ou non et ils peuvent aussi se partager des coprocesseurs. Un coprocesseur servant à effectuer des recherches dans la table de routage est souvent employé. Une unité de gestion des queues de paquets est un deuxième coprocesseur que l'on retrouve fréquemment. La gestion des queues de paquets permet de coordonner les processeurs qui accèdent aux paquets avec les interfaces d'entreés/sorties. Ce coprocesseur peut aussi effectuer la segmentation et le réassemblage des paquets ainsi que des contrôles pour assurer

une certaine qualité de service. Des NPU commerciaux comme le Motorola C-5e et le NP4GS4 de IBM suivent ces grandes lignes.

Dans ce type de plate-forme, les processeurs génèrent beaucoup de requêtes, que ce soit pour accéder au coprocesseur, à une mémoire partagée ou encore aux interfaces d'entrées/sorties. Tous ces accès passent par un réseau d'inters qui possède des délais intrinsèques et qui doit gérer la contention. En combinant cela aux délais encourus dans les coprocesseurs et les mémoires partagées, il en résulte que les processeurs de la plate-forme peuvent passer une partie considérable de leur temps à attendre sans rien faire d'utile. Tel que décrit à la section 1.2,un processeur HMT ou SMT peut aider à masquer cette latence en passant rapidement à un second processus lorsque le premier est bloqué sur une certaine opération.

L'avantage principal du processeur HMT par rapport aux autres méthodes de traitement multiprocessus est qu'il est simple à réaliser. Par exemple, avec un processeur RISC comme point de départ sur lequel on désire implémenter un ordonnanceur qui fonctionne en mode tourniquet, il n'est pas nécessaire d'ajouter beaucoup de matériel au processeur. Les principales modifications à faire seraient de dupliquer les registres de contrôle et de données afin de créer un ensemble de registres différents pour chaque processus supporté et d'utiliser un compteur et des multiplexeurs afin d'envoyer aux unités fonctionnelles du pipeline les informations provenant d'un ensemble de registres différents à chaque cycle. Ce type de processeur change donc de processus à tous les cycles et, lorsque le nombre de processus supportés est supérieur ou égal au nombre d'étages du pipeline, il est garanti qu'il n'y aura pas d'aléas de contrôle ou de données. La simplicité du processeur HMT explique pourquoi il figure parmi les premières techniques utilisées pour réaliser des superordinateurs dans les années 1980 [3, 28, 78]. Tel qu'expliqué à la section 1.3, avec l'arrivée des SoC, ce type de processeur a récemment regagné en popularité, particulièrement dans le domaine des processeurs réseaux. Dans [61], un modèle d'un NPU a été réalisé avec des processeurs HMT et les résultats montrent que cette approche offre des gains

substantiels.

4.2 Techniques de modélisation

Puisque l'utilisation d'un processeur HMT semble être intéressante pour les NPU, nous désirons disposer d'un modèle simulable afin d'inclure ce type de processeur dans notre environnement de développent à haut niveau. Les processeurs de type HMT, SMT ou CMP gagnent en popularité et plusieurs compagnies, incluant Tensilica, ARM et IBM, ont annoncé qu'elles travaillaient sur l'une ou l'autre de ces technologies. Malheureusement, en ce moment, il n'y a pas de modèle simulable (ISS) d'un processeur HMT existant qui puisse être obtenu pour notre recherche. Cette section se penchera donc sur des méthodes de modélisation d'un processeur HMT à partir d'un ISS d'un processeur RISC. Un tel modèle, bien qu'il ne représente pas un processeur réel, peut nous offrir des estimés de performances assez précis pour pouvoir évaluer la pertinence d'utiliser une telle technologie dans un design.

4.2.1 Utiliser des banques de registre

Utiliser des banques de registres est une approche qui modélise l'élément de base d'un processeur HMT, c'est-à-dire dupliquer tous les registres afin de pouvoir sauvegarder l'état de chaque processus. Tel qu'expliqué à la section 2.3.1, la structure générale d'un ISS est une boucle qui lit chacune des instructions une à la fois et qui exécute le comportement désiré. Pour ce faire, l'ISS utilise une série de variables qui représentent l'état complet du processeur (les registres de données, les registres de contrôles et des bits d'états). Pour modéliser un processeur supportant N processus, il suffit de dupliquer toutes ses variables N fois et, à chaque tour de la boucle, de changer la banque de registres qui est utilisée pour simuler la prochaine instruction. Lorsque le code source de l'ISS est disponible, cette modification est relativement simple à

réaliser.

Il est aussi possible d'utiliser un ordonnanceur plus complexe qu'un tourniquet. Par exemple, il est possible de noter quel processus est en attente sur un accès externe et quel processus est prêt à être exécuté. Quand vient le temps de changer de banque de registres, le choix s'effectue toujours en mode tourniquet, mais uniquement au niveau des processus prêts. Un autre mode d'ordonnancement est d'attribuer une priorité dynamique à chaque processus où celui qui est prêt à s'exécuter et qui a attendu le plus longtemps depuis sa dernière exécution devient prioritaire. Ce type d'ordonnancement matériel est évidemment simple à réaliser dans un modèle simulable, mais demande un peu plus de travail dans une réalisation matérielle. De plus, il est important de noter que ces ordonnancements, incluant le tourniquet, sont appropriés lorsque le nombre total de processus concurrents à exécuter correspond exactement au nombre de processus supportés par le matériel du processeur. Si le nombre total de processus logiciel à exécuter est plus grand, il est alors nécessaire de passer par un système d'exploitation pour gérer quel processus logiciel sera associé, pour une certaine durée, à un processus matériel.

Un dernier point qui doit être pris en considération est la cohérence des communications. En effet, lorsqu'un processus fait une requête qui doit passer par le NoC et revenir après un délai non déterministe, nous devons garantir que lorsque la réponse arrive, c'est le bon processus qui la reçoit. Heureusement, le protocole OCP (et donc SOCP par extension) supporte déjà un mécanisme à cet effet. Lorsque le processeur effectue une requête sur le canal SOCP, un numéro d'identification pour le processeur (`masterID`) et un autre pour le processus (`threadID`) sont envoyés. Lorsqu'une réponse arrive dans le module SystemC de l'ISS, il est donc possible de savoir à quel processus la réponse est destinée. Ce dernier peut alors être ajouté sur la liste des processus prêts à s'exécuter et il recevra ses données lorsque son tour viendra. L'ordonnanceur, qui s'exécute dans un SC_THREAD à chaque cycle d'horloge, s'occupe d'envoyer les bonnes données au bon processus. Si nécessaire, il est, bien entendu,

possible de garder la donnée reçue dans un tampon jusqu'à ce qu'elle soit consommée. La figure 4.1 illustre la structure générale d'un module SystemC fonctionnant selon cette méthode.

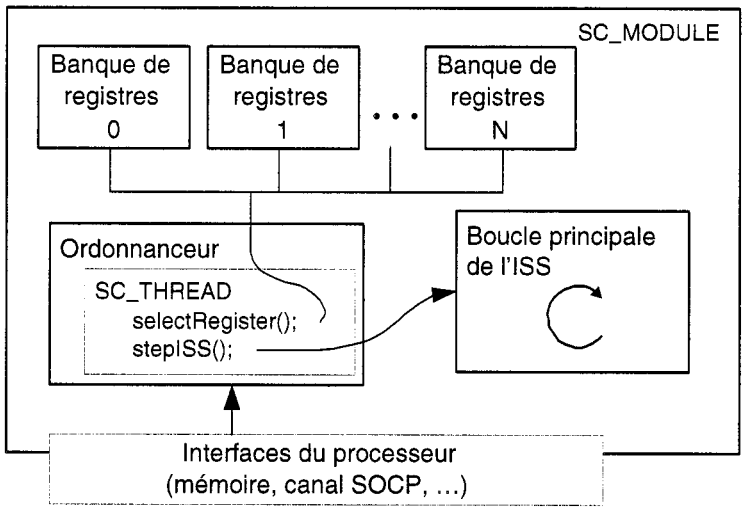


FIGURE 4.1 Processeur HMT utilisant plusieurs banques de registres

4.2.2 Utiliser plusieurs instances de l'ISS

Si l'on ne désire pas aller jouer dans le code de l'ISS, ou si les sources ne sont pas disponibles, il est possible d'utiliser plusieurs instances de l'ISS au lieu des banques de registres. Pour ce faire, il suffit de créer plusieurs SC_THREAD contenant chacun l'ISS et de les englober dans un module SystemC. Ce module contient aussi un SC_THREAD sensible à l'horloge globale de la plate-forme qui agit comme ordonnanceur. À chaque cycle, il décide laquelle des instances de l'ISS peut s'exécuter. La figure 4.2 montre la structure générale du module.

Dans ce cas-ci, il n'est pas nécessaire d'utiliser des banques de registres puisque chaque instance de l'ISS possède son propre ensemble de registres et chaque instance de l'ISS s'occupe d'un seul des processus concurrents. Chacun des ISS contient son propre SC_THREAD qui attend sur un événement de type sc_event. Lorsque l'or-

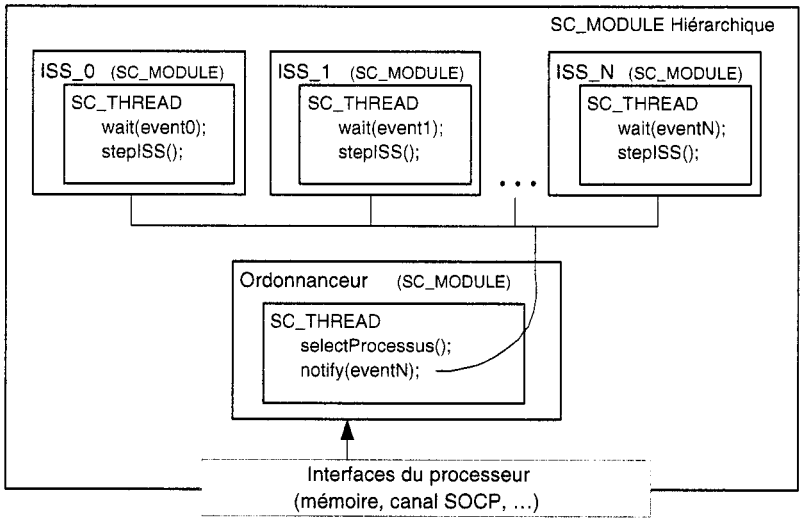


FIGURE 4.2 Processeur HMT utilisant plusieurs instances de l'ISS

donnanceur choisit le prochain processus qui sera exécuté, il envoie une notification à l'événement approprié. Comme c'est le cas avec la méthode des banques de registres, il est possible d'utiliser différentes techniques d'ordonnancement.

4.3 Le Xtensa avec support HMT

Afin de réaliser un modèle de processeur HMT, il a été décidé de se baser sur l'ISS du Xtensa. Cela permet de bénéficier des avantages des instructions spécialisées tout en ajoutant la possibilité de masquer efficacement la latence des communications. Les deux techniques de modélisation décrites précédemment ont été utilisées et certains détails spécifiques au Xtensa sont donnés dans cette section.

4.3.1 Banques de registres

Puisque le code source de l'ISS du Xtensa n'est pas disponible, il n'est pas possible d'aller dupliquer les registres directement. Cependant, l'interface XTMP de l'ISS offre des fonctions pour aller lire ou écrire directement dans les registres du processeur, ce

qui permet de contourner le problème. Il est alors possible de se créer une fonction pour aller lire l'ensemble des registres et les placer dans un tableau en mémoire, et une autre fonction pour écrire le contenu de ce tableau dans les registres de l'ISS. La figure 4.3 montre la boucle principale (l'ordonnanceur) qui change la banque de registres à chaque cycle.

```
void Xtensa_HMT::stepXtensa(void)
{
    bool cont = true;
    for (int i=0; i<nbThreads; i++) readAllReg(i);
    currentThread = 0;
    while(cont) {
        cont = stepIss(); // Exécute une instruction
        // Lire tous les registres qui viennent d'être modifiés
        readAllReg(currentThread);
        // Changer le processus (thread) courrant
        if (currentThread == nbThreads-1) currentThread = 0;
        else currentThread++;
        // Écrire dans l'ISS les registres du nouveau processus
        saveAllReg(currentThread);
    }
}
```

FIGURE 4.3 Ordonnanceur utilisant plusieurs banques de registres

Le problème majeur avec cette technique est qu'il est nécessaire de déplacer tous les registres du processeur deux fois par cycle de simulation (de l'ISS vers le tableau de sauvegarde et vice-versa). Cette tâche s'exécute en zéro cycle de simulation et ne fausse donc pas les résultats du profilage, mais elle demande beaucoup de travail de la part de la machine hôte de la simulation. Il en résulte que même une application triviale de quelque lignes demande plusieurs minutes pour être simulée. Pour une application de la taille de Click, les délais deviennent inacceptables et cette approche a donc dû être écartée.

4.3.2 Plusieurs instances de l'ISS

Cette seconde technique de modélisation d'un processeur HMT s'applique beaucoup mieux avec l'ISS du Xtensa. Le problème de performance rencontré avec les banques de registres est contourné puisque chaque instance de l'ISS possède ses propres registres et aucune copie n'est nécessaire. Une condition importante pour pouvoir utiliser plusieurs instances du même ISS est que ce dernier puisse être exécuté dans un environnement multiprocessus (comme SystemC) où les variables globales sont partagées. Un ISS qui utiliserait, par exemple, des variables globales pour modéliser ses registres ne pourrait pas fonctionner dans un tel environnement. L'ISS du Xtensa a été conçu dès le départ pour pouvoir s'exécuter dans un environnement multiprocessus et ne souffre pas de ce problème.

Chaque instance de l'ISS s'exécute dans une boucle nommée `runXtensaThread` (voir figure 4.4) qui est associée à un SC_THREAD. Cette fonction reçoit le numéro d'identification unique du processus qui s'exécutera sur le processeur et utilise une fonction qui permet d'exécuter une instruction à la fois (`stepIssHmt`). La boucle `runXtensaThread` n'est pas sensible à un signal d'horloge, elle se synchronise plutôt en attendant sur un événement généré par l'ordonnanceur. La figure 4.5 montre la structure utilisée pour sauvegarder l'état d'un processus. L'ordonnanceur se sert de cette structure pour envoyer le signal de départ et aussi pour noter le moment où un processus donné pourra s'exécuter à nouveau. L'ordonnanceur, présenté à la figure 4.6, fonctionne en mode tourniquet sur les processus qui sont prêts à s'exécuter uniquement. Si la majorité des processus sont en attente, alors les processus actifs bénéficient ainsi de plus de temps sur le processeur. Cependant, si un seul processus s'exécute à chaque cycle, il risque d'y avoir des aléas dans le pipeline du processeur ce qui élimine un des avantages d'un processeur HMT. C'est pour cela qu'il peut être intéressant pour un processeur HMT de limiter la fréquence à laquelle un processus donné peut lancer une instruction dans le pipeline. Pour ce faire, un paramètre

configurable (`pipeDepth`) est utilisé par l'ordonnanceur. Cela permet de s'assurer que deux instructions du même processus ne se retrouvent pas dans le pipeline en même temps. Si l'on ne désire pas modéliser ce comportement, il suffit de fixer le paramètre `pipeDepth` à 0.

```
void Xtensa_HMT::runXtensaThread(int tid)
{
  // Associé le numéro du SC_THREAD au numéro du processus (tid)
  tidMap[sc_get_curr_process_handle()->proc_id] = tid;
  while(true) {
      // Indiquer à l'ordonnanceur que le processus (thread) est prêt
      ts[tid].hwThreadState = ThreadState::READY;
      // Attente sur le signal de départ de l'ordonnanceur
      wait(ts[tid].threadEvent);
      ts[tid].hwThreadState = ThreadState::RUNNING;
      currentThread = tid;
      stepIssHmt(tid); // exécute 1 instruction
    }
}
```

FIGURE 4.4 Mécanisme de synchronisation d'un processus sur le Xtensa HMT

```
struct ThreadState {
  // signal sur lequel le processus attend
  sc_event threadEvent;
  // prochain moment où le processus peut s'exécuter
  unsigned long nextIssueTime;

  enum TSTATE {
      UNKNOWN,
      READY,   // le processus est prêt à s'exécuter
      RUNNING  // le processus s'exécuter déjà
  }
  hwThreadState;
};
```

FIGURE 4.5 Structure utilisée pour sauvegarder l'état d'un processus

4.3.3 Support pour le logiciel

Un point important n'a pas été abordé jusqu'à maintenant : la manière dont le logiciel est exécuté sur le processeur HMT. Dans un environnement multiprocessus, que ce

```
void Xtensa_HMT::stepXtensa(void)
{
  currentThread = 0;
  int i,j;
  unsigned nextChoice = 0;
  while(!stopSim) {
    gotOne = false;
    for(i=0; i<nbThreads; i++) {
      j = (i + nextChoice) % nbThreads;
      ThreadState &cs = ts[j];  // Lire l'état du processus j
      if( (cs.hwThreadState == ThreadState::READY) &&
          (issueTime >= cs.nextIssueTime)) {
        // Calculer le prochain moment où le processus pourra s'exécuter
        cs.nextIssueTime = issueTime + pipeDepth;
        cs.threadEvent.notify(); // Débloque le processus sélectionné
        if(++nextChoice >= nbThreads) nextChoice = 0;
        gotOne = true;
        break;
      }
    }
    issueTime++;
    wait();
  }
}
```

FIGURE 4.6 Ordonnanceur utilisant plusieurs instances de l'ISS

soit avec un processeur HMT ou plusieurs processeurs séparés, la manière dont le logiciel est supporté revêt une grande importance. Afin de faciliter le développement d'une application dans notre environnement sans système d'exploitation nous nous sommes fixés les objectifs suivants :

- Tous les processus devraient être capables d'exécuter le même code ou des applications différentes sans problème.
- Afin de facilement partager des données entre deux processus, la même plage d'adresses devrait être accessible par tous les processus. De cette manière, il est possible d'imaginer un scénario où un processus envoie une adresse à un second processus qui peut alors accéder aux données qui lui sont destinées.
- Si deux processus exécutent le même code, ils devraient utiliser le même espace mémoire pour les instructions et les données globales, mais posséder une pile privée (pour les données locales).

En ce qui a trait au premier point, puisque dans notre cas le processeur HMT est simulé avec plusieurs instances de l'ISS, il est facile d'exécuter une application différente sur chacun des processus. Cependant, la plage d'adresses où le Xtensa va chercher ses instructions est définie lors de la création du processeur. Si chaque instance de l'ISS utilise exactement le même processeur, ils vont donc par défaut aller lire le même programme. Ce problème se contourne facilement en modifiant la plage d'adresses utilisée par chaque instance de l'ISS au début de la simulation.

Afin de faciliter l'échange de données, tous nos processeurs sont configurés sans unité de gestion de la mémoire et n'utilisent donc pas d'adresses virtuelles. Cela implique que chaque processeur voit l'ensemble de la plage d'adresses de 32 bits. Si nous voulions utiliser un système d'exploitation tel Linux ou VxWorks, il serait alors nécessaire d'utiliser des adresses virtuelles, mais ce n'est pas le cas avec Click.

Le troisième objectif demande un peu plus de travail pour être atteint. La portion de droite de la figure 4.7, montre la structure d'un programme en mémoire. Lorsqu'un fichier est compilé, il est divisé en différentes sections (ou segments).

- Segment texte : la portion de l'application qui contient les instructions à exécuter.
- Segment données : la portion qui contient les variables globales initialisées.
- BSS (*Block Started by Symbol*) : cette section contient des données globales non initialisées (identifiées uniquement par un nom et une taille).
- Mémoire dynamique (*heap*) : région utilisée lorsque des blocs de mémoires sont alloués en cours d'exécution.
- Pile (*stack*) : région utilisée pour sauvegarder les données locales à une fonction.

Lorsqu'une fonction est exécutée, toutes les variables locales sont poussées sur la pile puis retirées quand la fonction se termine, cela permet de facilement supporter des fonctions imbriquées. Si deux processus utilisent le même espace mémoire pour leur pile, des conflits majeurs vont inévitablement se produire puisque les données considérées locales et privées dans le code de l'application vont alors être partagées. Il est donc nécessaire de s'assurer que deux processus exécutant le même code sur notre

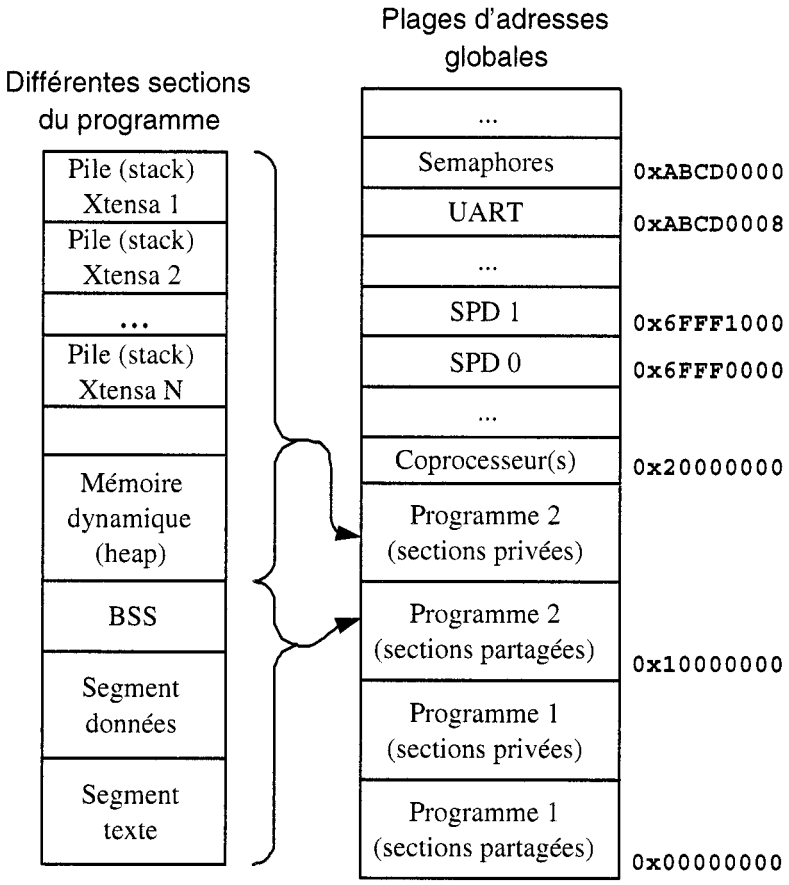


FIGURE 4.7 Plages d'adresses et structure d'un programme en mémoire

processeur HMT aient des piles séparées. Pour la gestion de la mémoire dynamique, la fonction `malloc()` de la bibliothèque du langage C est déjà capable de gérer l'allocation et la libération d'un espace mémoire partagé par plusieurs processus légers. Toutes les autres portions du programme sont partagées, cela permet à deux processus de s'échanger des messages au travers d'une variable globale. De plus, le fait de partager le segment texte, qui est toujours accédé en lecture uniquement, permet de sauver de l'espace mémoire. Lorsque des processus partagent un sous-ensemble de la mémoire, comme c'est le cas ici, ils sont habituellement nommés processus légers (ou *thread* en anglais).

La méthode la plus simple pour s'assurer que chacun des processus du processeur HMT disposent d'une pile privée est d'ajouter un dispositif logiciel qui permet de fixer la valeur du registre qui pointe sur la pile (le registre SP) lorsque le processus est initialement lancé. Lorsqu'un système d'exploitation est employé, c'est lui qui assure cette fonctionnalité. La figure 4.8 montre le code ajouté pour modifier le pointeur SP sans utiliser de système d'exploitation. Les 3 premières lignes définissent une variable globale qui contient une adresse mémoire (0xABCD0000). Dans l'ISS du Xtensa, cette petite plage d'adresses spéciale est interceptée et permet d'effectuer quelques opérations spécifiques pour la simulation. En plus d'obtenir l'adresse de la pile privée d'un processus, il est possible d'arrêter l'exécution du programme ou encore de simuler des affichages à l'écran en faisant des accès à cette plage d'adresses. La fonction principale (`main()`) du programme commence par faire une lecture à cette adresse et utilise le résultat pour modifier SP. Lorsque l'ISS reçoit une lecture à cette adresse, il ne la transfert pas à une mémoire. Il commence plutôt par détecter quel processus a fait cet appel, puis il répond directement en envoyant l'adresse de départ de la pile du processus. Après cela, un saut inconditionnel est effectué dans la fonction `realMain()` et l'exécution normale du programme débute. Puisque tous les processus reçoivent une adresse différente de la part de l'ISS, ils disposent effectivement d'une pile privée.

```
asm(".align 4\n"
    ".threadStack:\n\t"
    ".word 2882338816"); // adresse spéciale : 0xABDC0000

main() {
    asm("l32r a4, .threadStack\n\t" // Lecture interceptée par l'ISS
        "l32i sp, a4, 0\n\t"
        "j realMain");
}

void realMain(void)
{
    // L'application débute ici...
}
```

FIGURE 4.8 Code assembleur utilisé pour attribuer des piles privées

4.3.4 Coprocesseur pour le sémaphores

Lorsque plusieurs processus peuvent s'échanger des données à l'aide de variables partagées, il est nécessaire de fournir un mécanisme pour garantir qu'il n'y aura pas de conflits (comme deux écritures simultanées par exemple). Dans un système d'exploitation, le mécanisme des sémaphores est habituellement employé pour garantir à un processus l'exclusivité à une ressource. Sur notre plate-forme, il est possible d'utiliser un coprocesseur partagé entre tous les processeurs pour gérer les sémaphores.

Lorsque l'application est créée, chaque ressource partagée est associée à un sémaphore. Dans le coprocesseur de sémaphores, chacun des différents sémaphores correspond à une adresse. Un processus qui veut accéder à une ressource partagée commence par faire une lecture à l'adresse associée au sémaphore désiré. Cette lecture est reçue par le coprocesseur de sémaphores qui regarde alors la disponibilité de la ressource désirée. Si le processus reçoit une valeur différente de zéro, il peut accéder à la ressource. Lorsque le processus a fini son traitement, il relâche le sémaphore en écrivant un zéro à l'adresse appropriée. Puisque dans un canal de communication OCP chaque processus est identifiable uniquement grâce au `masterID` et au `threadID`, le coprocesseur de sémaphores peut noter quel processus possède quel sémaphore et s'assurer que c'est effectivement le bon processus qui relâche le sémaphore.

Si le sémaphore n'est pas disponible, deux approches sont possibles. Dans le premier cas, la lecture du sémaphore n'est pas bloquante et, s'il n'est pas disponible, le processus reçoit un zéro comme réponse. Il peut alors essayer de nouveau jusqu'à ce qu'il obtienne le sémaphore en utilisant un mécanisme de scrutation. La deuxième méthode est d'utiliser des accès bloquants, c'est-à-dire que si le sémaphore n'est pas disponible, aucune réponse n'est envoyée au processus. De cette manière, le processus est bloqué jusqu'à ce qu'il reçoive une réponse positive. Pour simplifier le code de l'application, c'est ce deuxième mécanisme qui est utilisé dans notre plate-forme.

L'emploi d'un coprocesseur de sémaphores s'avère utile pour l'affichage de texte à l'écran. Dans un SoC, le code exécuté n'effectue habituellement pas d'écritures sur un écran pour la simple et bonne raison qu'aucun dispositif d'affichage n'est disponible. Cependant, lors de la phase de développement, cela s'avère extrêmement utile pour voir les résultats produits par l'application et s'assurer que tout fonctionne correctement. C'est pourquoi l'ISS du Xtensa supporte la fonction d'affichage standard `printf()` et redirige ces appels vers la machine hôte de la simulation. La fonction `printf()` demande plusieurs cycles pour s'exécuter, même si le message affiché est très court. Dans un processeur HMT, il va, de tout évidence, y avoir un changement de processus avant la fin de l'affichage. Si plus d'un processus exécute une écriture à l'écran en même temps, les chaînes de caractères vont apparaître emmêlées les une dans les autres. Pour régler ce problème, un autre coprocesseur, cette fois-ci pour l'affichage, est ajouté. Pusique l'utilité du coprocesseur d'affichage se limite principalement à la simulation, il est situé dans la plage d'adresses spéciale utilisée pour la simulation et décrite précédemment (section 4.3.3). Ce coprocesseur exécute une seule fonction très simple, il affiche à l'écran le caractère envoyé par un des processus. La figure 4.9, montre le code d'une nouvelle fonction ayant la même interface que `printf()`. Cette fonction copie la chaîne de caractères à afficher dans un tampon, elle demande le sémaphore associé au coprocesseur d'affichage et finalement envoie les caractères un à la fois. Cela garantit que le texte envoyé par un processus s'affiche complètement avant que le texte d'un autre processus débute. Notons finalement que malgré que l'utilité de ce coprocesseur se limite surtout à la simulation, son fonctionnement est très semblable à celui d'un module d'entrées/sorties sériel UART (*Universal Asynchronous Receiver/Transmitter*) qui peut être utilisé pour envoyer des caractères sur un terminal à partir d'un circuit.

```
#define XT_SEM_OSPRINT 2
#define XT_MAGIC_UART  0xABCD0008
int tprintf(char *format, ...)
{
    int i;
    va_list ap;
    char buf[5000];
    va_start(ap, format);
    vsprintf(buf,format,ap);
    va_end(ap);
    i = MAGIC_SEM(XT_SEM_OSPRINT);  // demande le sémaphore
    // écriture de tous les caractères à l'adresse du UART
    for(i = 0; buf[i] != 0; i++)
       ACCESS_MM(XT_MAGIC_UART) = (int) buf[i];
    MAGIC_SEM(XT_SEM_OSPRINT) = 0;  // relâche le sémaphore
    return 0;
}
```

FIGURE 4.9 Fonction d'impression à l'écran utilisant un sémaphore

4.3.5 Implémentation matérielle

L'objectif principal derrière le modèle simulable d'un processeur Xtensa avec support pour le HMT est de pouvoir vérifier quels sont les gains qu'un processeur de ce type peut offrir. Le premier but à atteindre n'est donc pas de modéliser en SystemC un processeur qui existe déjà en matériel. Il est cependant bon de s'assurer que le modèle SystemC créé peut réalistement être implémenté en matériel.

Tel que décrit à la section 4.2.1, la technique qui consiste à dupliquer les registres du processeur est conceptuellement simple à réaliser si le code source Verilog ou VHDL du processeur est disponible. Cependant, comprendre la grande quantité de code d'un processeur et ensuite le modifier intelligemment peut demander beaucoup de temps. La seconde technique de modélisation employée (plusieurs instances de l'ISS) peut sembler à première vue plus loin d'un équivalent matériel que la première technique, mais ce n'est pas le cas en réalité.

La figure 4.10 présente le schéma d'une implémentation matérielle qui s'approche beaucoup de notre modèle de processeur Xtensa HMT. Chaque instance de l'ISS

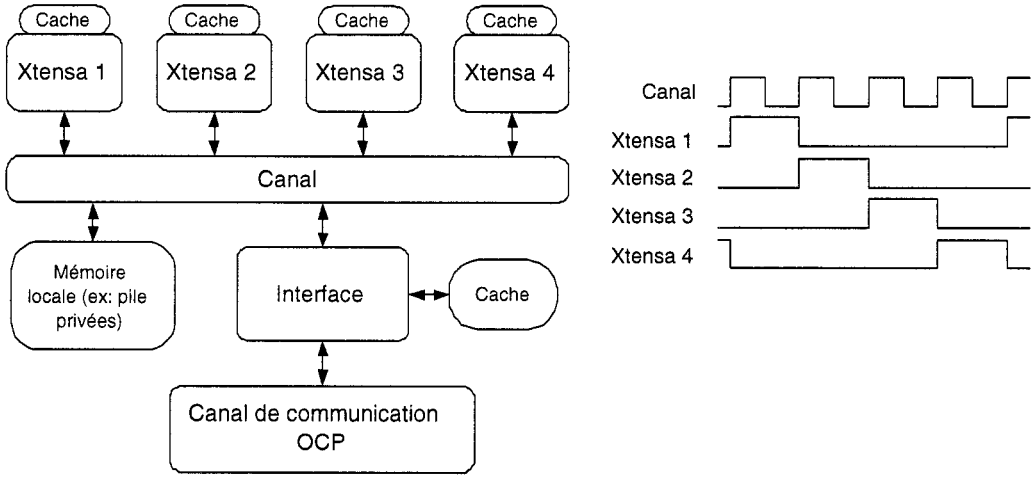


FIGURE 4.10 Implémentation matérielle d'un processeur HMT

représente un processeur Xtensa complet. Tous les processeurs sont reliés entre eux par un canal privé, c'est-à-dire un canal qui est isolé du reste de la plateforme. Ce canal fonctionne à une fréquence plus élevée que chacun des processeur (4 fois plus élevée dans le cas de la figure 4.10 puisqu'il y a 4 processeurs). De plus, les 4 horloges lentes reliées aux processeurs sont déphasées de manière à ce que un seul des processeurs effectue une requête sur le canal pour chacun de ses cycles. Ce déphasage dans les horloges implique donc un ordonnancement de type tourniquet entre chacun des processeurs. Le paramètre `pipeDepth` décrit à la section 4.3.2 peut donc être vu comme un moyen de modéliser le fait qu'un processeur peut lancer une nouvelle instruction uniquement une fois à tous les 4 cycles du canal.

L'interface entre le groupe des 4 processeurs et le reste de la plate-forme est responsable de transférer les appels des processeurs vers le canal de communication OCP. De plus, lorsqu'une réponse revient, l'interface doit l'entreposer jusqu'à ce que le cycle correspondant au processeur visé soit atteint. Pour le reste de la plate-forme, qui ne voit que l'interface, ce bloc se comporte donc comme un processeur HMT supportant 4 processus. Les autres blocs (les caches et la mémoire) de la figure 4.10 sont optionnels. Par exemple, il est possible que chacun des processeurs individuels aient une mémoire cache privée. Une mémoire cache partagée entre tous les processeurs

du groupe peut aussi être imaginée. Cette mémoire peut, par exemple, être utilisée comme cache de niveau 2 pour les instructions (si chaque processeur exécute le même code) et être connectée à l'interface du module. Il est aussi envisageable d'inclure une mémoire partagée entre tous les processeurs du groupe si l'application exécutée demande beaucoup de partage de données entre les processeurs.

Il est donc clair que les deux techniques de modélisation d'un processeur HMT décrites à la section 4.2 sont réalisables en matériel. Dupliquer les banques de registre permet d'obtenir un processeur plus compact puisque, contrairement à la solution de la figure 4.10, il n'y a qu'un seul pipeline qui est partagé entre tous les processeurs. Cependant, utiliser plusieurs processeurs fonctionnant à une fréquence réduite offre l'avantage qu'il n'est pas nécessaire de modifier un processeur existant. Un autre étudiant de notre groupe de recherche se penche présentement sur l'implémentation d'un processeur HMT de ce type dans un FPGA [36].

4.4 Plate-forme de simulation simple

Afin d'obtenir des premiers résultats et de valider le fonctionnement du processeur Xtensa supportant le HMT, une première plate-forme de simulation simple, mais néanmoins réaliste, est construite. Cette plate-forme est présentée à la figure 4.11, et chacun des éléments de cette plate-forme ainsi que les paramètres configurables et l'application exécutée sont par la suite décrits.

4.4.1 Éléments de la plate-forme

Chacun des processeurs Xtensa utilisé dans le module HMT est identique et ses options de configuration sont les mêmes que celles données dans le tableau 3.3 à une exception prêt : la taille des mémoires caches peut être modifiée. Par exemple, un

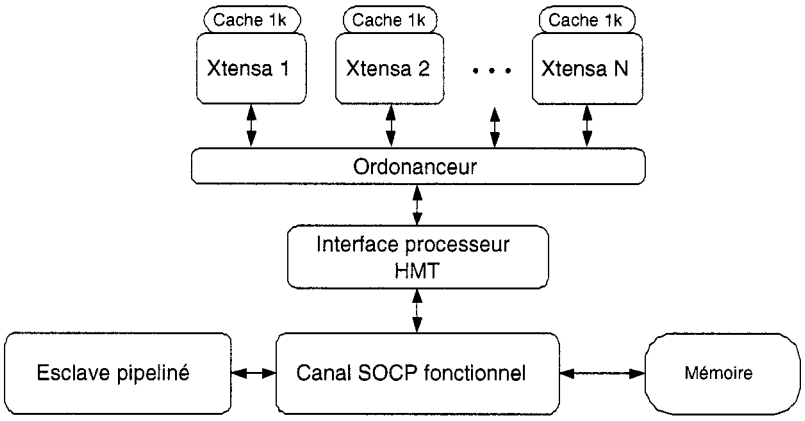


FIGURE 4.11 Plate-forme de simulation simple pour un processeur HMT

premier test peut être effectué sans aucune mémoire cache. Dans ce cas, tous les accès mémoires des processeurs doivent passer par le canal de communication pour atteindre la mémoire principale. Cela génère beaucoup de trafic sur le canal et permet de mesurer les pertes de performances créées pas les délais des communications. La plate-forme peut aussi être configurée pour utiliser des processeurs qui disposent chacun d'une mémoire cache d'instructions ou de données (ou les deux). Dans ce cas, chaque processus peut-être vu comme disposant de sa propre mémoire cache privée.

Le canal de communication utilisé ici est un canal TLM comme celui utilisé pour la plate-forme de simulation des applications IPv4 et IPSec (figure 3.1). La seule différence est que, dans ce cas-ci, une certaine latence est modélisée dans le canal. Modéliser les délais dans un NoC de manière précise demande un modèle assez détaillé d'un NoC existant. Cela est nécessaire pour pouvoir modéliser la contention, les blocages, le temps de traitement dans le NoC et les délais des inters. Puisque modéliser un Noc avec un haut niveau de détails n'est pas le but de cette recherche, un modèle très simple a été utilisé. La latence est représentée comme un délai (en nanosecondes) lorsqu'un maître envoie une requête à un esclave. Une fois la requête traitée, la réponse de l'esclave revient sans aucun délai. Le délai ajouté dans le canal (sous la forme d'une commande `wait()` de SystemC) peut donc être vu comme la

latence moyenne pour tous les transferts de données sur le canal.

Le dernier composant de la plate-forme est l'esclave. Cet esclave peut, par exemple, représenter un module d'entrées/sorties, un coprocesseur pour effectuer une recherche dans une table de routage (dans le cas d'une application comme IPv4) ou encore un coprocesseur calculant une transformée en cosinus discret (dans le cas d'une application d'encodage vidéo). Dans le cas de notre plate-forme simple, l'esclave permet de modéliser une ressource partagée entre les différents processeurs. Il ne réalise donc pas une fonction complexe, mais s'occupe simplement de répondre au maître après avoir simulé un certain un temps de calcul. L'esclave, illustré à la figure 4.12, possède une file d'attente pour les requêtes en entrées et une autre pour les requêtes en sorties. Le fonctionnement de l'esclave est très simple puisque, tel que mentionné précédemment, il n'effectue aucune fonction complexe. Lors d'une écriture, la donnée est sauvegardée dans une mémoire interne et lors d'une lecture, cette donnée est retournée après avoir été additionnée avec une valeur constante arbitraire. Puisque l'interface SOCP fournit à l'esclave un numéro d'identification unique pour chaque processus (threadID) et pour chaque maître (masterID), les données lues et écrites par chacun des processus du processeur Xtensa HMT peuvent être séparées dans des espaces mémoires différents. Notons finalement que l'esclave modélise un pipeline, ce qui lui permet d'émettre une réponse à chaque cycle. Dans ces conditions, le nombre de cycles requis pour qu'un maître reçoive une réponse dépend de la quantité de requêtes déjà dans la file d'attente et du nombre de cycles requis par l'esclave pour traiter la requête.

4.4.2 Paramètres configurables

L'objectif de la plate-forme de simulation simple est de mesurer les gains offerts par un processeur HMT sous différentes conditions. Pour ce faire un certain nombre de paramètres sont modifiables. Pour le canal de communication, le seul paramètre

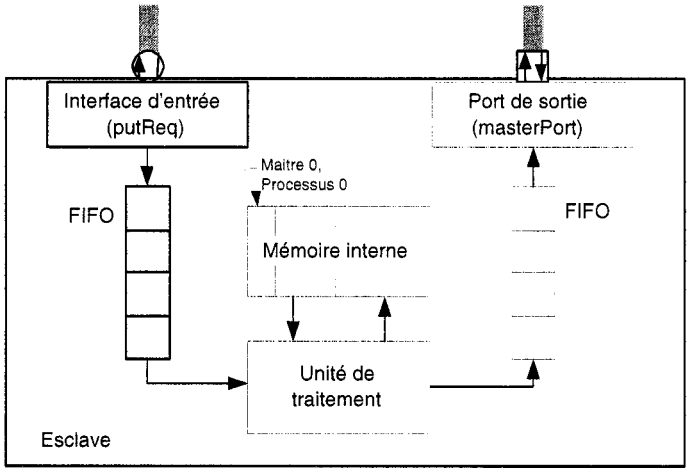


FIGURE 4.12 Esclave utilisé dans la plate-forme simple

configurable est la latence moyenne (en nanoseconde) imposée pour chacune des transactions. Lorsque le processeur effectue un accès mémoire en rafale, le temps d'attente est égal à la latence moyenne du canal plus un cycle de délai (trois nanoseconde dans notre cas puisque le processeur est simulé avec une horloge ayant un période de 3 ns) par mots dans l'accès mémoire en rafale. Du côté du processeur, il est possible de modifier le nombre de processus supportés (de 1 à 32) et la taille de la mémoire cache d'instructions et de données (0 à 16 ko). Finalement, sur l'esclave, les paramètres configurables sont la profondeur du pipeline, qui détermine le temps de traitement d'une écriture et d'une lecture (en nombre de cycles), ainsi que la taille de la file d'attente en entrée et en sortie.

4.4.3 Application exécutée

Une application simple s'exécute sur la plate-forme de la figure 4.11. Tout d'abord, chaque processus effectue plusieurs écritures dans l'esclave à différentes adresses. Par la suite, les données écrites sont relues et une vérification est faite pour s'assurer que l'esclave les a modifiées correctement (la tâche de l'esclave est d'additionner une constante à la valeur écrite). De plus, des opérations de bases (additions et divisions)

sont effectuées sur les données lues afin de simuler du temps de traitement sur le processeur. La simplicité de cette application a l'avantage de permettre des temps de simulations courts tout en se comportant comme une application plus complexe (Click, par exemple) qui effectue un grand nombre de transactions sur le canal de communication.

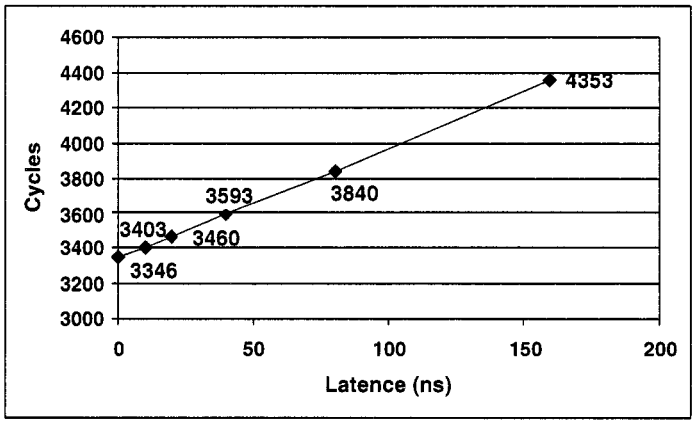
4.5 Résultats

Différentes simulations ont été effectuées avec l'application IPv4 présentée au chapitre 3 ainsi qu'avec la plate-forme simple présentée à la section 4.4. Les résultats obtenus sont présentés dans cette section. Par la suite, les limitations des simulations effectuées ainsi que les avantages des processeurs HMT sont abordés.

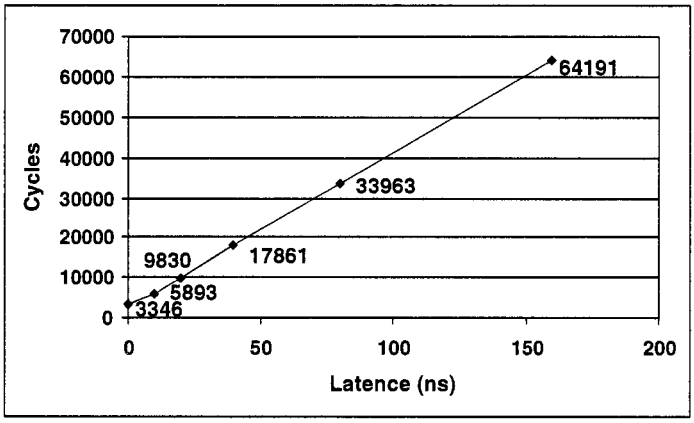
4.5.1 Impact de la latence des communications sur l'application IPv4

Afin de démontrer l'impact négatif que peut avoir la latence des communications sur les performances de notre processeur réseau utilisant un Xtensa, plusieurs simulations ont été faites en modifiant à chaque fois la latence du canal et de la mémoire. L'application utilisée est IPv4 avec des paquets de taille minimale et le modèle de délais est le même que celui décrit à la section 4.4. C'est-à-dire que la latence du canal est modélisée comme un délai dans une seule direction et pour la mémoire, dans le cas particulier des accès en rafale, un délai additionnel d'un cycle par mots de 32 bits est ajouté. La figure 4.13(a) montre l'impact de la latence lorsque la mémoire est connectée directement au processeur. Dans ce cas, les accès à la mémoire se font toujours en un cycle alors que les accès aux modules d'entrée/sortie SPD doivent passer à travers le canal de communication et sont donc soumis à un temps d'attente. Tel qu'illustré sur le graphique de la figure 4.13(a), le temps de traitement d'un seul paquet de taille minimale (en nombre de cycles) augmente linéairement lorsque la

latence augmente. Si l'on trace une droite entre les différents points du graphique, la pente obtenue est d'environ 6 cycles par nanoseconde.



(a) Sur le canal uniquement



(b) Sur le canal et la mémoire

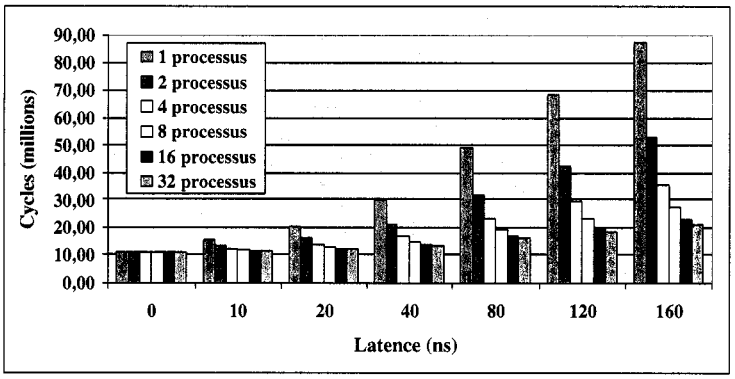
FIGURE 4.13 Impacte de la latence sur l'application IPv4

Le graphique présenté à la figure 4.13(b) montre quant à lui l'effet de la latence s'il y a un délai pour accéder au SPD et à la mémoire. Puisque dans l'application IPv4 simulée ici il n'y a pas de coprocesseur DMA pour accélérer les accès à la mémoire, seule la mémoire cache, qui est de 8 ko dans cet exemple, offre un temps d'accès rapide (1 cycle) au processeur. Tel qu'observé sur le graphique, le nombre de cycles nécessaire pour traiter un paquet augmente toujours linéairement avec la latence, mais de manière beaucoup plus rapide, la pente de la droite état d'environ 375 cycles par nanoseconde. C'est donc dire que dans ces simulations, lorsque la latence est non

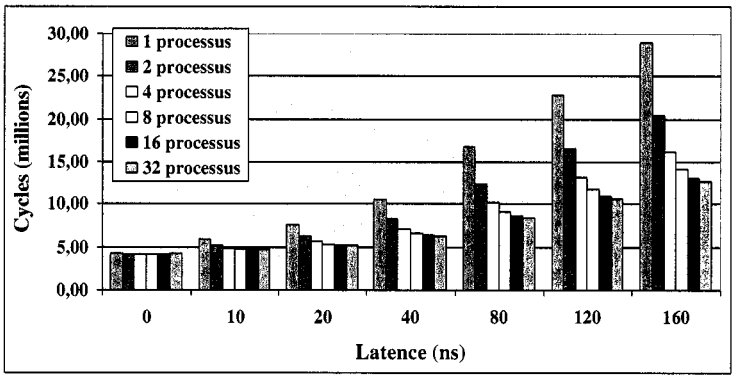
négligeable, le processeur passe la majorité de son temps à attendre pour accéder à la mémoire ou au SPD.

4.5.2 Gains obtenus avec la plate-forme de simulation simple

L'objectif de ces tests est de montrer à l'aide d'un exemple simple, les avantages d'un processeur HMT. Le premier ensemble de simulations effectuées sur la plate-forme simple fait varier la latence du canal de communication ainsi que le nombre de processus supportés par le processeur HMT. L'application exécutée doit échanger 128 ko de données avec l'esclave (64 ko en écriture et 64 ko en lecture) avant de se terminer. Lorsque le processeur HMT supporte plus d'un processus concurrent, chaque processus est responsable de transférer une fraction de ces 128 ko. Il s'agit donc d'une application qui, tout comme le traitement de paquets, est facile à paralléliser. Dans ce premier ensemble de simulations, le processeur ne possède aucune mémoire cache et il est donc obligé d'accéder à la mémoire principale très fréquemment. Le temps de traitement de l'esclave est fixé à 5 cycles et la taille de ses files d'attente est infinie. Avec un modèle de communication TLM (comme SOCP), si le module implémentant une interface (putReq() dans le cas de l'esclave), n'est pas capable de répondre immédiatement à la requête, il est nécessaire de mettre cette dernière dans une file d'attente. En SystemC, une interface peut, en effet, recevoir plusieurs requêtes dans le même cycle de simulation. Il est donc utile d'avoir des files infinies afin de ne pas perdre aucune transaction. Bien entendu, il est possible de vérifier la longueur de ces files en cours de simulation afin de s'assurer qu'un esclave ne reçoit pas une quantité ridicule de requêtes dans un court laps de temps. Dans une plate-forme monoprocesseur comme celle utilisée ici, l'esclave ne peut cependant pas recevoir plus d'une requête par cycle. L'histogramme de la figure 4.14(a), montre les performances obtenues lorsque la latence et le nombre de processus varient. La mesure de performance est le nombre de cycles requis pour échanger les 128 ko de données avec l'esclave.



(a) Aucune mémoire cache



(b) 1 ko de cache par processus

FIGURE 4.14 Performances obtenues avec un processeur HMT

Ces résultats montrent clairement que plus la latence est élevée, plus le processeur HMT offre des gains intéressants. La seconde conclusion à tirer est que le fait d'augmenter le nombre de processus supportés passé un certain point n'offre plus vraiment de gains substantiels. Par exemple, même avec une latence élevée de 120 ou 160 ns, le fait de passer de 8 à 32 processus offre seulement un gain de 1.3 qui est beaucoup plus petit que celui obtenu par le passage de 1 à 8 processus (gain de 3.2).

Le deuxième ensemble de simulations effectuées sur la plate-forme simple fait varier la latence du canal de communication ainsi que le nombre de processus supportés par le processeur HMT, mais cette fois-ci chaque processus sur le Xtensa HMT possède sa propre mémoire cache de 1 ko. Tel qu'illustré à la figure 4.14(b), cela améliore grandement les performances puisque la mémoire cache permet au processeur d'éviter

de faire un grand nombre d'accès sur le canal de communication. Une autre conclusion à tirer de ce graphique est que lorsque le nombre d'accès sur le canal diminue, le gain offert par un processeur HMT diminue aussi. Par exemple, avec un processeur sans cache et une latence de 40 ns, le fait de passer de 1 processus à 8 processus nous permet de doubler les performances (de 29,8 millions de cycles à 14,8). Lorsque chacun des processus possède sa mémoire cache le gain est plutôt de 1,5 (de 10,4 millions de cycles à 6,5).

D'autres simulations ont été faites avec des mémoires caches plus grandes sur chacun des processus. Avec des mémoires de 8 ko, le gain offert par le processeur HMT devenait beaucoup moins intéressant et cela même lorsque la latence du canal était très élevée. Cela vient confirmer qu'un processeur HMT est surtout utile lorsqu'un grand nombre d'accès doit être fait sur un canal lent. Il est aussi important de préciser que le processeur HMT masque la latence du canal et permet d'augmenter le débit (quantité de données traitées par secondes), mais il ne fait pas diminuer la latence. Dans le cadre d'un NPU, il devient donc possible de traiter plus de paquets par secondes, mais chaque paquet pris individuellement demande potentiellement plus de temps avant de sortir du routeur.

4.5.3 Améliorations possibles

Bien que les premiers résultats obtenus sont encourageants, plusieurs autres simulations devraient cependant encore être faites afin d'obtenir une meilleure idée des avantages et inconvénients d'un processeur HMT. Tout d'abord, il faudrait simuler une application Click complète sur un processeur HMT, par exemple l'application IPv4. Cela permettrait de calculer les gains obtenus par rapport aux résultats présentés à la section 4.5.1. Une première tentative infructueuse pour effectuer cette simulation a eu lieu. Afin de faire fonctionner Click sur un processeur HMT, différents obstacles doivent être surmontés. Par exemple, il est nécessaire d'utiliser une tech-

nique légèrement différente de celle de la figure 4.8 pour que chaque processus ait sa pile privée, car Click est un logiciel en C++ (et non en C) et le compilateur C++ du Xtensa n'est pas complètement compatible avec le standard C++. Cependant, le principal problème est que pour ce genre de simulation, l'ISS du Xtensa n'est pas utilisé tel que prévu. Lorsque l'ISS plante, un message d'erreur interne est retourné et aucune documentation ni code source n'est disponible pour éclaircir le problème. Tous les problèmes sont cependant contournables et c'est dû au manque de temps que la simulation de Click sur un processeur Xtensa HMT a été abandonnée pour ce mémoire. Cependant, le problème reste sous investigation et en ce moment nous essayons de modifier la version légère de Click (NanoClick, voir section 3.3.1) pour qu'elle s'exécute sur un processeur Xtensa HMT.

Une deuxième amélioration serait d'effectuer des simulations avec plusieurs processeurs et plusieurs périphériques différents. Ce type de plate-forme est typiquement retrouvé dans les NPU existant et tend à générer beaucoup de trafic sur le réseau d'inters. Il s'agirait donc d'un bon environnement pour tester des processeurs HMT puisque sur ce type de plate-forme les gains offerts par un processeur HMT seraient probablement significatifs même lorsque des mémoires cache sont utilisées.

Finalement, un modèle de mémoire cache plus réaliste serait nécessaire. Avec l'ISS du Xtensa il est impossible de rediriger les appels vers la mémoire cache dans le modèle SystemC. Nous nous retrouvons donc à modéliser un processeur HMT où chaque processus possède sa propre mémoire cache privée. Dans bien des cas, cette configuration n'est pas celle désirée. Il serait donc utile de créer notre propre modèle de mémoire cache, qui pourrait être attaché à l'interface du processeur tel qu'illustré à la figure 4.10. D'une manière plus générale, plusieurs simulations pourraient être réalisées afin de trouver, pour une application donnée, un compromis idéal entre la taille de la mémoire cache, le type d'organisation des caches, le nombre de processeurs et le nombre de processus concurrents sur chacun des processeurs.

CONCLUSION

La conception d'un système embarqué réalisé sous forme d'un système-sur-puce est devenue une tâche complexe. Les coûts de conception et de développement de ces circuits ne cessent d'augmenter et pour palier à ces problèmes plusieurs se tournent vers de nouvelles méthodes qui permettent d'augmenter le niveau d'abstraction. Un des objectifs est d'obtenir rapidement une plate-forme simulable afin de pouvoir, dans un premier temps, mesurer rapidement les impacts de certaines modifications architecturales et, dans un second temps, commencer le développement du logiciel dès le début de la conception du SoC. Le logiciel occupe d'ailleurs une place de plus en plus importante dans les nouveaux SoC développés. Cela donne une plate-forme flexible et peut permettre de créer différents produits en modifiant simplement le logiciel ce qui aide à absorber les coûts de production du circuit.

La toile de fond de ce mémoire porte sur la phase d'exploration architecturale lors de la conception d'un nouveau SoC. Lors de cette phase, le modèle simulable est construit à l'aide de la bibliothèque SystemC 2.0 qui est devenu le langage le plus populaire pour la spécification de systèmes logiciels/matériels. Afin de faciliter la conception et l'intégration de modules SystemC ensemble, l'outil de développement StepNP a été sélectionné. StepNP offre une bibliothèque de composants SystemC partageant une interface commune ainsi que des modèles de processeurs basés sur des simulateurs de jeux d'instructions (ISS). De plus, StepNP offre une série d'outils de débogage et d'analyse de performance qui viennent faciliter le développement d'une plate-forme avec SystemC.

Le premier des deux objectifs principaux de ce travail était d'évaluer, à haut niveau, les bénéfices offerts par l'utilisation de processeurs configurables. Le processeur Xtensa qui offre la possibilité de facilement ajouter des nouvelles instructions spécialisées au processeur a été sélectionné. Afin de mieux exploiter les possibilités de

ce type de processeurs, certaines modifications ont été apportées à la méthodologie de partitionnement logiciel/matériel classique. Ces modifications visent à favoriser l'utilisation d'instructions spécialisées face aux coprocesseurs. En effet, ces instructions spécialisées offrent dans bien des cas des gains de performance similaires aux coprocesseurs tout en demandant beaucoup moins d'effort pour la conception et l'implémentation. Avec le processeur Xtensa, la nouvelle instruction est décrite à l'aide d'un langage spécifique développé pour ce processeur. Par la suite, l'intégration du matériel nécessaire dans le processeur se fait automatiquement ce qui minimise le risque d'erreurs d'implémentation. De plus, aucune communication supplémentaire n'est nécessaire avec une instruction spécialisée, ce qui n'est pas le cas avec un coprocesseur.

Une fois cette méthodologie établie, l'ISS du Xtensa a été intégré dans StepNP afin de pouvoir développer des architectures à haut niveau utilisant des processeurs configurables. Pour valider la méthodologie et mesurer les gains offerts par les processeurs configurables, une plate-forme de simulation ainsi que des applications ont dû être créées. La création de la plate-forme monoprocesseur à l'aide de StepNP n'a posé aucun problème. Le domaine des processeurs réseaux a, quant à lui, été choisi pour la création des applications. Pour la réalisation du logiciel en tant que tel, l'environnement de développement de routeurs modulaires Click a été utilisé. Grâce à Click, deux applications réseaux représentatives ont rapidement été créées. La première application porte sur le routage des paquets selon le protocole IPv4 qui est en ce moment le plus répandu sur Internet. La seconde application développée est une extension de la première, elle ajoute le cryptage des paquets selon le protocole IPsec. Avec ces deux applications en main, la méthodologie proposée a été appliquée et des accélérations intéressantes ont rapidement été obtenues grâce à l'utilisation d'instructions spécialisées.

Le deuxième objectif principal de ce travail était d'évaluer, toujours à haut niveau, les bénéfices offerts par l'utilisation d'un processeur supportant plusieurs proces-

sus concurrents et des changements de contexte rapides. Un processeur réseau doit traiter rapidement une grande quantité d'informations. Pour ce faire, il est souvent nécessaire d'utiliser plusieurs processeurs afin de traiter plusieurs paquets en parallèle. Or, ce faisant un grande quantité d'information doit alors transiter sur le réseau d'interconnections qui ajoute souvent une latence non négligeable sur le transfert des données. Si aucun mécanisme n'est utilisé pour masquer cette latence, les processeurs vont alors passer la majorité de leur temps en attente. Cela réduit beaucoup les gains offerts par l'ajout de nouveaux processeurs. Le processeur HMT est une solution permettant de masquer la latence qui est simple à implémenter et qui a déjà fait ses preuves dans le passé.

En se basant sur l'ISS du Xtensa, un modèle de processeur HMT a été implémenté. Cela permet de bénéficier des avantages des instructions spécialisées tout en étant capable d'efficacement masquer la latence des communications. Pour valider ce modèle de processeur HMT, une plate-forme de simulation a été créée avec StepNP. Dans cette plate-forme, il est possible de contrôler plusieurs paramètres dont le nombre de processus supportés par le processeur HMT et la latence du canal de communication. Cela nous a permis d'obtenir rapidement une bonne idée des gains offerts par un processeur HMT.

Principaux résultats

Tout d'abord nous avons pu démontrer que l'ajout d'instructions spécialisées peut accélérer significativement certaines portions d'une application complexe. Cela s'avère particulièrement marquant si l'on regarde le cas de l'algorithme de chiffrement DES. Le principal avantage des instructions spécialisées reste cependant la facilité avec laquelle elles peuvent être créées et intégrées dans le processeur et dans le code de l'application originale. Cependant, même si les instructions spécialisées sont avantageuses, l'utilisation d'un coprocesseur demeure parfois incontournable. Par exemple,

un coprocesseur peut effectuer des déplacements de paquets en mémoire pendant que le processeur effectue d'autres traitements. Ce type de parallélisme n'est pas possible avec une instruction spécialisée.

En second lieu, l'efficacité d'un processeur HMT a été démontrée. Il a été montré que ce type de processeur offre des gains de vitesse marqués surtout lorsqu'un grand nombre d'accès est fait sur un canal de communication avec une forte latence. Cependant, même lorsqu'il y a peu de latence à masquer, un processeur HMT reste un moyen simple et peu coûteux (comparativement à un processeur superscalaire par exemple) d'aller chercher une augmentation des performances. De plus, il a été montré que l'efficacité du processeur HMT est plus marquée lorsque des petites mémoires caches sont utilisées, ce type de processeur peut donc être utilisé pour compenser le manque de mémoire cache (la cache peut être réduite ou éliminée pour des considérations de surface du circuit par exemple).

Travaux futurs

Plusieurs autres travaux pourraient découler de cette recherche. La prochaine étape serait d'apporter certaines améliorations évidentes tant du côté du processeur réseau créé que du côté du modèle de processeur HMT. Tout d'abord, le logiciel Click vise à être modulaire et flexible avant d'être performant. Il en résulte que l'application demande beaucoup de temps d'initialisation et que ces performances ne sont pas toujours optimales. Il serait préférable d'utiliser la version allégée de Click décrite à la section 3.3.1. De plus, les deux applications utilisées comportaient certaines simplifications dont une petite table de routage, l'absence de fragmentation des paquets et l'absence d'authentification lors d'une communication encryptée. Si l'on regarde la plate-forme d'un processeur réseau qui a été utilisée, il serait intéressant d'étudier d'autres facteurs pouvant affecter les performances, par exemple la taille de la mémoire cache ou encore la largeur des bus du processeur. Cependant, la

modification la plus évidente pour améliorer les performances reste l'utilisatio de plusieurs processeurs. Ajouter plus d'un processeur dans l'environnement StepNP est très simple à réaliser, mais par la suite il, serait nécessaire d'adapter l'application pour partager de manière efficace le travail entre les processeurs.

En ce qui a trait au modèle de processeur HMT, il serait intéressant d'ajouter un modèle de mémoire cache créé en SystemC afin de mieux mesurer les impacts d'une cache partagée entre plusieurs processus concurrents sur les performances du système. Avec l'ISS du Xtensa, il est impossible de rediriger les appels vers la mémoire cache dans notre module SystemC, comme cela est fait avec les mémoires. La seule manière de modéliser une cache partagée est donc de configurer un Xtensa sans cache interne et par la suite de la modéliser nous-même en SystemC. Une autre amélioration à envisager serait d'effectuer des simulations plus réalistes avec des processeurs HMT. Par exemple, il serait intéressant de faire rouler l'application IPSec sur une plate-forme multiprocesseur. Dans cet environnement une très grande quantité de trafic transigerait sur le NoC et l'avantage du processeur HMT pour masquer la latence des communications apparaîtrait encore plus clairement.

Si l'on regarde dans une perspective plus globale les travaux qui peuvent découler de ce travail, une première voie prometteuse à investiguer est la génération automatique d'instructions spécialisées. Dans notre cas, toutes les instructions ont été construites manuellement après inspection de l'application. Il existe cependant des nouvelles techniques de génération automatique qui semblent prometteuses [4]. Ces techniques apparaissent déjà dans des nouveaux produits comme le processeur Stetch (voir annexe I.3). Il serait pertinent de comparer ces techniques automatiques à une génération manuelle afin d'évaluer les avantages et les inconvénients des deux approches. Tensilica a d'ailleurs annoncé la sortie prochaine (fin 2004) d'une technologie nommée XPRESS qui va permettre la génération automatique d'instructions TIE.

L'utilisation du Xtensa a causé certains problèmes. Il s'agit d'un excellent produit, probablement le plus mature sur le marché en ce moment, et s'il est utilisé tel que proposé par Tensilica, alors les outils et la documentation fournie fonctionnent très bien. Cependant, dans ce travail nous avons voulu intégrer l'ISS du Tensilica dans un environnement de développement déjà existant (StepNP) au lieu de se servir de la bibliothèque XTMP du Xtensa comme environnement de développement. Nous avons aussi voulu modéliser un processeur HMT à partir du Xtensa bien qu'un tel processeur n'existe actuellement pas. Dans ces deux cas, l'absence du code source de l'ISS et l'absence de documentation sur le fonctionnement interne du simulateur nous a forcé à investir beaucoup d'efforts pour utiliser le simulateur d'une manière non prévue par Tensilica. Pour contourner ces problèmes et aussi pour étudier d'autres options il serait intéressant d'utiliser d'autres types de processeurs configurables. L'outil LISATek, décrit dans la section 1.1.2, semble être un excellent choix dans ce domaine.

Finalement, lorsque l'on effectue de l'exploration architecturale, il est toujours important de penser au raffinement de la plate-forme développée vers une implémentation physique. Plus précisément, il serait utile de disposer d'un processeur HMT au niveau RTL afin d'implémenter notre processeur réseau dans un FPGA et effectuer des tests réels. Le code RTL du Xtensa n'est pas disponible avec notre licence universitaire, il faut donc se tourner vers d'autres voies. Encore une fois LISATek semble être une option intéressante. Il existe aussi un processeur RISC, nommé OpenRISC[1], dont le code RTL est disponible gratuitement. De plus, ce processeur possède déjà une option pour supporter plusieurs processus concurrents.

[1]http ://www.opencores.org

RÉFÉRENCES

[1] ACE. *The COSY compilation system.* Associated Compiler Experts, http ://www.ace.nl/products/cosy.html, 2004. (Page consultée le 17 septembre 2004).

[2] AGARWAL, A., LIM, B., KRANZ, D., AND KUBIATOWICZ., J. *APRIL : a processor architecture for multiprocessing.* 17th Annual International Symposium on Computer Architecture, Mai 1990.

[3] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. *The Tera Computer System.* Internationnal Conference on Supercomputing, Juin 1990.

[4] ATASU, K., POZZI, L., AND LENNE, P. *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints.* Design Automation Conference (DAC), 2003.

[5] AUSTIN, T. *SimpleScalar Hacker's Guide.* SimpleScalar LLC, http : www.simplescalar.com, 2004. (Page consultée le 17 septembre 2004).

[6] AUSTIN, T., LARSON, E., AND ERNST, D. *SimpleScalar : An Infrastructure for Computer System Modeling.* IEEE, Février 2002.

[7] BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. *Metropolis : An Integrated Electronic System Design Environment.* IEEE Computer, 2003.

[8] BENINI, L., BERTOZZI, D., BRUNI, D., DRAGO, N., FUMMI, F., AND PONCINO, M. *SystemC Cosimulation and Emulation of Multiprocessor SoC Designs*, vol. 36. IEEE Computer, Avril 2003.

[9] BENNY, O. *Implémentation d'un modèle de communication transactionnel dans une plate-forme en SystemC.* École Polytechnique de Montréal. Mémoire de Maîtrise, 2004.

[10] Bois, G., Filion, L., Tsikhanovich, A., and Aboulhamid, E. *Modélisation, raffinement et techniques de programmation orientée objet avec SystemC.* Publication en cours, 2004.

[11] Braden, R., and Borman, D. *Computing the Internet Checksum (RFC 1071).* http ://www.ietf.org/rfc.html, 1988. (Page consultée le 17 septembre 2004).

[12] Brooks, D., Tiwari, V., and Martonosi, M. *Wattch : a framework for architectural-level power analysis and optimizations.* 27th International Symposium on Computer Architecture, 2000.

[13] Cesário, W. O., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A. A., Gauthier, L., and Diaz-Nava, M. *Multiprocessor SoC Platforms : A Component-Based Design Approach.* IEEE Design and Test of Computers, Novembre 2002.

[14] Chevalier, J., Rondonneau, M., Benny, O., Bois, G., Aboulhamid, E.-M., and Boyer, F.-R. *SPACE : A Hardware/Software SystemC modeling platform including an RTOS.* Forum on specification & Design Languages (FDL), 2003.

[15] Claasen, T. A. *System on a Chip : Changing IC Design Today and in the Future.* IEEE Micro, Mai - juin 2003.

[16] Crowley, P., Fiuczynski, M. E., Baer, J.-L., and Bershad, B. N. *Characterizing Processor Architectures for Programmable Network Interfaces.* 2000 International Conference on Supercomputing, 2000.

[17] Cyr, G. *Interface configurable pour un processeur ARM basée sur le protocole VCI.* Mémoire de Maîtrise. École Polytechnique de Montréal, Février 2001.

[18] Deslauriers, F. *Création d'un environnement de test pour l'évaluation de performance de réseaux.* École Polytechnique de Montréal. Mémoire de Maîtrise en préparation, 2004.

[19] DUTT, N., AND CHOI, K. *Configurable processors for embedded computing.* Computer, Volume : 36 , Issue : 1, Janvier 2003.

[20] EGGERS, S. J., AND AL. *Simultaneous Multithreading : A Platform for Next-Generation Processors*, vol. 17. IEEE Micro, 1997.

[21] EZER, G. *Xtensa with User Defined DSP Coprocessor Microarchitectures.* IEEE, 2000.

[22] FILION, L. *Analyse, implantation et intégration d'une bibliothèque pour la spécification des systèmes embarqués dans une méthodologie de codesign.* École Polytechnique de Montréal. Mémoire de Maîtrise, 2002.

[23] FUMMI, F., MARTINI, S., PERBELLINI, G., AND PONCINO, M. *Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC.* Design, Automation and Test in Europe (DATE), 2004.

[24] GAJSKI, D. D., ZHU, J., DÖMER, R., AND GERSTLAUER, A. *SpecC : Specification Language and Methodology.* Kluwer Academic Publishers, Norwell, MA, 2000.

[25] GONZALEZ, R. E. *Xtensa : A configurable and Extensible Processor.* IEEE, Juin 2000.

[26] GRIES, M. *Methods for Evaluating and Covering the Design Space during Early Design Development.* Technical report, Electronics Research Lab, University of California, Berkeley, UCB/ERL M03/32, Août 2003.

[27] GRÖTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. *System Design with SystemC.* Kluwer Academic Publishers, Mai 2002.

[28] HALSTEAD, R., AND FUJITA, T. *MASA : A multithreaded processor architecture for parallel symbolic computing.* 15th Annual International Symposium on Computer Architecture, Mai 1988.

[29] HAMMOND, L., HUBBERT, B. A., SIU, M., PRABHU, M. K., CHEN, M., AND OLUKOTUN, K. *The Stanford Hydra CMP*, vol. 20. IEEE Micro, 2000.

[30] HAMMOND, L., NAYFEH, B. A., AND OLUKOTUN, K. *A Singe-Chip Multiprocessor.* IEEE, 1997.

[31] HAVERINEN, A., LECLERCQ, M., WEYRICH, N., AND WINGARD, D. *White Paper for SystemC based SoC Communication Modeling for the OCP Protocol.* Open SystemC Initiative, http ://www.systemc.org, Octobre 2002. (Page consultée le 17 septembre 2004).

[32] HENKEL, J. *Closing the SoC Design Gap.* Computer, Septembre 2003.

[33] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture a Quantitative Approach*, second ed. Morgan Kaufmann Publishers, San Francisco, 1990.

[34] HOFFMANN, A., KOGEL, T., NOHL, A., BRAUN, G., SCHLIEBUSCH, O., WAHLEN, O., WIEFERINK, A., AND MEYR, H. *A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language*, vol. Volume 20, Issue 11. Computer-Aided Design of Integrated Circuits and Systems, Novembre 2001.

[35] HOUNSELL, B., AND TAYLOR, R. *Co-processor synthesis : a new methodology for embedded software acceleration.* Design, Automation and Test in Europe Conference and Exhibition, Février 2004.

[36] HUBIN, M. *Conception et implantation d'une architecture de traitement multiprocessus matériel.* École Polytechnique de Montréal. Mémoire de Maîtrise en préparation, 2004.

[37] HYLANDS, C., LEE, E., LIU, J., LIU, X., NEUENDORFFER, S., XIONG, Y., ZHAO, Y., AND ZHENG, H. *Overview of the Ptolemy Project.* Technical Memorandum UCB/ERL M03/25, EECS, University of California Berkeley, Juillet 2003.

[38] JAY LO, J. L. *Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors.* Thèse de Doctorat. Université de Washington, 1998.

[39] KALLA, R., SINHAROY, B., AND TENDLER, J. *IBM power5 chip : a dual-core multithreaded processor*, vol. 24. Micro, IEEE, Mars 2004.

[40] KENT, S., AND ATKINSON, R. *IP Authentication Header (RFC 2402).* http ://www.ietf.org/rfc.html, 1998. (Page consultée le 17 septembre 2004).

[41] KENT, S., AND ATKINSON, R. *IP Encapsulating Security Payload (RFC 2406).* http ://www.ietf.org/rfc.html, 1998. (Page consultée le 17 septembre 2004).

[42] KENT, S., AND ATKINSON, R. *Security Architecture for the Internet Protocol (RFC 2401).* http ://www.ietf.org/rfc.html, 1998. (Page consultée le 17 septembre 2004).

[43] KEPPEL, D. *Tools and Techniques for Building Fast Portable Threads Packages.* University of Washington, rapport technique UWCSE 93-05-06, ftp ://ftp.cs.washington.edu/tr/1993/05, 1993. (Page consultée le 17 septembre 2004).

[44] KEUTZER, K., MALIK, S., NEWTON, R., RABAEY, J. M., AND SANGIOVANNI-VINCENTELLI, A. *System-Level Design : Orthogonalization of Concerns and Platform-Based Design.* IEEE, 2000.

[45] KOHLER, E. *The Click Modular Router.* Thèse de doctorat. Massachusetts Institute of Technology., http ://www.pdos.lcs.mit.edu/click, Février 2001. (Page consultée le 17 septembre 2004).

[46] LYONNARD, D., YOO, S., BAGHDADI, A., AND JERRAYA, A. A. *Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip.* Design Automation Conference (DAC), 2001.

[47] MAGARSHACK, P., AND PAULIN, P. G. *System-on-Chip Beyong the Nanometer Wall.* Design Automation Conference, DAC'03, Juin 2003.

[48] MARR, D. T., AND AL. *Hyper-Threading Technology Architecture and Microarchitecture.* Intel Technology Journal (Q1), 2002.

[49] MCDOWELL, L. K., EGGERS, S. J., AND GRIBBLE, S. D. *Improving Server Software Support for Simultaneous Multithreaded Processors.* Principles and Practice of Parallel Programming, Juin 2003.

[50] MIHAL, A., AND KEUTZER, K. *Mapping Concurrent Applications onto Architectural Platforms.* A. Jantsch, H. Tenhunen, 3, 39-59, Kluwer Academic Publishers, 2003.

[51] MIHAL, A., KULKARNI, C., MOSKEWICZ, M., TSAI, M., SHAH, N., WEBER, S., JIN, Y., KEUTZER, K., SAUER, C., VISSERS, K., AND MALIK, S. *Developing Architectural Platforms : A Disciplined Approach.* IEEE Design and Test of Computers, Novembre 2002.

[52] MONG, W. S., AND ZHU, J. *A Retargetable Micro-architecture Simulator.* Design Automation Conference (DAC), 2003.

[53] MOSHOVOS, A., AND SOHI, G. *Microarchitectural innovations : boosting microprocessor performance beyond semiconductor technology scaling.* Proceedings of the IEEE , Volume : 89 , Issue : 11, Novembre 2001.

[54] NIST FIPS. *Data Encryption Standard (DES).* National Institute of Standards and Technology, http ://www.itl.nist.gov/fipspubs/fip46-2.htm, 1993. (Page consultée le 17 septembre 2004).

[55] OCP-IP ASSOCIATION. *Open Core Protocol Specification.* OCP International Partnership Association, http ://www.ocpip.org, 2001. (Page consultée le 17 septembre 2004).

[56] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. *The Case for a Single-Chip Multiprocessor.* ASPLOS, 1996.

[57] OSCI. *Functional Specification for SystemC 2.0.1.* Open SystemC Initiative, http ://www.systemc.org, 2002. (Page consultée le 17 septembre 2004).

[58] OSCI. *SystemC Version 2.0.1 User's Guide.* Open SystemC Initiative, http ://www.systemc.org, 2002. (Page consultée le 17 septembre 2004).

[59] OSCI. *SystemC 2.0.1 Language Reference Manual Revision 1.0.* Open SystemC Initiative, http ://www.systemc.org, 2003. (Page consultée le 17 septembre 2004).

[60] PAULIN, P. G., PILKINGTON, C., AND BENSOUDANE, E. *StepNP : A System-Level Exploration Platform for Network Processors.* IEEE Design and Test of Computers, Novembre 2002.

[61] PAULIN, P. G., PILKINGTON, C., AND BENSOUDANE, E. *Network processing challenges and an experimental NPU platform.* Design, Automation and Test in Europe, 2003.

[62] PAULIN, P. G., PILKINGTON, C., BENSOUDANE, E., LANGEVIN, M., AND LYONNARD, D. *Application of a multi-processor SoC platform to high-speed packet forwarding.* Design, Automation and Test in Europe, 2004.

[63] PAULIN, P. G., PILKINGTON, C., LANGEVIN, M., BENSOUDANE, E., SZABO, K., AND LYONNARD, D. *A Multi-Processor SoC Platform and Tools for Communications Applications*, embedded systems handbook ed. CRC Press, Janvier 2005.

[64] PEES, S., HOFFMANN, A., AND MEYR, H. *Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language.* Design, Automation and Test in Europe (DATE), 2000.

[65] PEES, S., HOFFMANN, A., ZIVOJNOVIC, V., AND MEYR, H. *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures.* Design Automation Conference, New Orleans, June 1999.

[66] PEES, S., ZIVOJNOVIC, V., HOFFMANN, A., AND MEYR, H. *Retargetable Timed Instruction Set Simulator of Pipelined Processor Architectures.* International Conference on Signal Processing Applications and Technology (ICSPAT), Toronto, Septembre 1998.

[67] PRICE, C. *MIPS IV Instruction Set, revision 3.1.* MIPS Technologies, Inc., Mountain View, CA,, Janvier 1995.

[68] QUINN, D. *Exploration architecturale pour la conception de processeurs réseaux basée sur l'utilisation de processeurs configurables.* École Polytechnique de Montréal. Mémoire de Maîtrise, 2003.

[69] QUINN, D., LAVIGUEUR, B., BOIS, G., AND ABOULHAMID, M. *A System Level Exploration Platform and Methodology for Network Applications Based on Configurable Processors.* Design Automation and Test in Europe, DATE'04, Février 2004.

[70] RIJSINGHANI, A. *Computing of the Internet Checksum via Incremental Update (RFC 1624).* http ://www.ietf.org/rfc.html, 1994. (Page consultée le 17 septembre 2004).

[71] RONDONNEAU, M. *Intégration d'un RTOS dans une plate-forme SystemC destinée à l'exploration architecturale.* École Polytechnique de Montréal. Mémoire de Maîtrise, 2003.

[72] SANGHAVI, J., AND WANG, A. *Estimation of Speed, Area, and Power of Parametrizable, Soft IP.* Design Automation Conference, Juin 2001.

[73] SCHAUMONT, P., VERBAUWHEDE, I., KEUTZER, K., AND SARRAFZADEH, M. *A Quick Safari through the Reconfigurable Jungle.* Design Automation Conference, 2001.

[74] SCHLIEBUSCH, O., HOFFMANN, A., NOHL, A., BRAUN, G., AND MEYR, H. *Architecture Implementation Using the Machine Description Language LISA.* 15th International Conference on VLSI Design (VLSID.02), 2002.

[75] SHAH, N. *Understanding Network Processors.* Mémoire de Maîtrise. University of California, Berkeley, 2001.

[76] SHAH, N., AND KEUTZER, K. *Network Processors : Origin of Species.* The Seventeenth International Symposium on Computer and Information Sciences, 2002.

[77] SHAH, N., PLISHKER, W., AND KEUTZER, K. *NP-Click : A Programming Model for the Intel IXP1200.* 2nd Workshop on Network Processors at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, Février 2003.

[78] SMITH, B. J. *Architecture and applications of the HEP multiprocessor computer system.* SPIE Real Time Signal Processing IV, 1981.

[79] SMITH, J., AND SOHI, G. *The microarchitecture of superscalar processors*, vol. 83 of *12.* Proceedings of the IEEE , Volume : 83 , Issue : 12, Décembre 1995.

[80] STORINO, S., AIPPERSPACH, A., BORKENHAGEN, J., EICKEMEYER, R., KUNKEL, S., LEVENSTEIN, S., AND UHLMANN, G. *A commercial multithreaded RISC processor.* International Solid-State Circuits Conference, Février 1998.

[81] SUN MICROSYSTEMS. *UltraSPARC IV Processor Architecture Overview.* Technical Whitepaper Version 1.0, Sun Microsystems, http ://www.sun.com, 2004. (Page consultée le 17 septembre 2004).

[82] SWAN, S. *An Introduction to System Level Modeling in SystemC 2.0.* Open SystemC Initiative, http ://www.systemc.org, 2001. (Page consultée le 17 septembre 2004).

[83] TENDLER, J., DODSON, J., FIELDS, J., LE, H., AND SINHAROY, B. *POWER4 System Microarchitecture.* IBM Journal of Research and Development, 46(1), 2002.

[84] TOMASEVIC, M., AND MILUTINOVIC, V. *A Simulation Study of Snoopy Cache Coherence Protocols.* Hawaii International Conference on System Sciences, 1992.

[85] TREMBLAY, M., CHAN, J., CHAUDHRY, S., CONIGLIARO, A. W., AND TSE, S. S. *The Majc Architecture : A Synthesis Of Parallelism And Scalability.* IEEE Micro, 2000.

[86] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. *Simultaneous Multithreading : Maximizing On-Chip Parallelism.* International Symposium on Computer Architecture, 1995.

[87] VIBHATAVANIJ, K., TZENG, N., AND KONGMUNVATTANA, A. *Simultaneous Multithreading-Based Routers.* IEEE, 2000.

[88] WANG, A., KILLIAN, E., MAYDAN, D., AND ROWEN, C. *Hardware/Software Instruction Set Configurability for System-on-Chip Processors.* Design Automation Conference (DAC), 2001.

[89] WATHEQ EL-KHARASHI, M., ELGUIBALY, F., AND LI, K. *Multithreaded processors : the upcoming generation for multimedia chips.* Advances in Digital Filtering and Signal Processing, Juin 1998.

[90] XIE, H., ZHAO, L., AND BHUYAN, L. *Architectural Analysis and Instruction-Set Optimization for Design of Network Protocol Processors.* CODES+ISSS, 2003.

[91] ZHANG, X., AND QIN, X. *Performance Prediction and Evaluation of Parallel Processing on a NUMA Multiprocessor*, vol. 17 of *10.* IEEE Transactions on Software Engineering, 1991.

[92] ZIVOJNOVIC, V., PEES, S., AND MEYR, H. *LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design.* IEEE Workshop on VLSI Signal Processing, San Francisco, Octobre 1996.

ANNEXE I

AUTRES PROCESSEURS CONFIGURABLES

I.1 ARC Tangent

Les processeurs Tangent de la compagnie ARC International[1] se partagent avec les Xtensa le marché des processeurs embarqués configurables. Tout comme le Xtensa et contrairement à des processeurs fixes tels les MIPS ou les ARM, les processeurs Tangent-A5 et A4 offrent certaines options de configuration lors de la création du processeur. Si l'on compare le processeur Tangent-A5 et le Xtensa les similitudes sont nombreuses. Le A5 est à la base un processeur RISC avec un pipeline de 4 étages. Ce processeur utilise un jeu d'instructions avec des instructions encodées sur 16 ou 32 bits afin de réduire la taille du code. Tout comme avec le Xtensa il est aussi possible de greffer des capacités DSP (principalement des multiplicateurs, des accumulateurs et des instructions SIMD) sur le A5. Il possède aussi une interface graphique permettant de sélectionner les options désirées sur le processeur. Par exemple, les options suivantes sont disponibles :

- la mémoire cache d'instructions et de données (associative par ensemble de 1, 2 ou 4 avec une taille de 0 à 32 ko) ;
- le nombre de registres généraux ;
- l'ajout d'un multiplicateur 32 bits ;
- le nombre et types d'interruptions et de compteurs ;
- des modules externes pour communication sur Ethernet, USB ou UART.

Il s'agit aussi d'un processeur fournis sous forme de code RTL synthétisable et intégrable dans un SoC. De plus, ARC fournit la chaîne complète d'outils logi-

[1] http ://www.arc.com

ciels pour développer des programmes sur le A5. La principale différence entre le ARCtengent-A5 et le Xtensa est que le A5 n'offre pas à l'usager les outils pour facilement modifier lui-même le jeu d'instructions du processeur et en étendre les capacités comme c'est le cas pour le Xtensa, ce qui le rend moins intéressant.

I.2 Cascade de Critical Blue

La compagnie écossaise Critical Blue offre depuis quelques mois un produit nommé Cascade qui permet d'automatiser la génération de co-processeurs dans un système sur puce[2]. La méthodologie utilisée par Cascade pour optimiser une application est semblable à celle du Xtensa et commence avec le code de l'application embarquée déjà existante. La première étape consiste à profiler le code C ou C++ existant afin de déterminer quelles sont les boucles qui demandent le plus de temps à s'exécuter. Par la suite, un coprocesseur optimisé pour ces boucles est automatiquement généré et connecté au processeur principal.

Une des particularités de Cascade est qu'il effetue son travail directement sur le code compilé (le microcode) de l'application. Cela rend l'outil indépendant du compilateur utilisé et même du langage utilisé, mais cela implique aussi que Cascade doit être adapté pour chaque nouveau processeur puisque chaque processeur possède un jeu d'instructions différent. C'est donc à partir du microcode que Cascade décide automatiquement (en tenant compte de paramètres définissables par l'usager) quelles parties devraient être réalisées par un coprocesseur. Les coprocesseurs de Cascade possèdent tous un ensemble d'unités fonctionnelles de base tels une interface avec le bus, une mémoire cache d'instruction et de donnée, un décodeur d'instruction et un contrôleur de pipeline. Après avoir analysé le code de l'application, d'autres unités utiles sont ajoutées au coprocesseur afin d'extraire le maximum de parallélisme des

[2]http ://www.criticalblue.com

boucles à optimiser [35].

Le coprocesseur utilise le même microcode que le processeur principal afin de déterminer ses tâches à effectuer. Il donc nécessaire d'avoir un seul code exécutable pour le système complet. Pour les boucles qui seront affectées au coprocesseur, le microcode original est optimisé afin d'aller chercher de meilleures performances. Le fait que le coprocesseur utilise le même code que le processeur permet une intégration transparente au niveau de l'application. Le coprocesseur est directement attaché au processeur à l'aide de l'interface bus de ce dernier et d'une mémoire partagée. Le coprocesseur est accessible comme une mémoire avec une plage d'adresses spécifiques et le microcode du processeur est modifié afin d'insérer les appels nécessaires pour transférer une fonction originellement effectuer sur le processeur vers le coprocesseur. Plus précisément, les 3 premières instructions de l'appel de fonction sont modifiés afin d'effectuer le branchement. Le coprocesseur possède sa propre mémoire cache qui contient le microcode à effectuer. Cela donne une certaine flexibilité au coprocesseur puisqu'il peut être reprogrammé.

En plus de générer automatiquement du code RTL pour le coprocesseur, Cascade fournit aussi un modèle en C précis au niveau des cycles d'horloge du coprocesseur. Cela permet de facilement vérifier si la fonctionnalité est respectée et de simuler le coprocesseur avec le reste de la plate-forme dans un environnement de simulation tel SystemC.

Pour l'instant Cascade crée des coprocesseurs uniquement à partir de code compilé pour un ARM7 ou ARM9, mais le support pour d'autres processeurs est en cours de développement. Cet outil très intéressant offre donc un niveau d'automatisation plus élevé que le Xtensa mais possède certaines limites. De plus, le surcoût causé par les communications et la synchronisation avec le processeur n'est pas très bien documenté à ce jours.

I.3 Stretch

La nouvelle compagnie Stretch Inc.[3] vise à offrir, elle aussi, une solution qui permet d'automatiquement augmenter les performances d'un processeur pour une application donnée. Le processeur S5000 développé par cette compagnie utilise un processeur Xtensa de Tensilica combiné avec de la logique programmable semblable à celle d'un FPGA. La force de ce processeur est son compilateur C qui détecte lui-même les points chauds à optimiser dans le code. Ce compilateur est aussi capable de produire des instructions TIE utiles pour optimiser le code et de générer le nécessaire pour que ces instructions TIE soient créées dans la logique configurable du processeur. Le Xtensa et la logique programmable communiquent entre eux à l'aide d'un banque de registres de 128bits afin d'offrir une grande bande passante. Il en résulte donc un processeur qui est capable de s'adapter au code exécuté. Contrairement au Xtensa seul, le S5000 est un circuit physique et n'est pas destiné à être inclus dans un système sur puce. Stretch vise commencer à vendre son processeur au courant de l'été 2004.

I.4 Improv Systems Jazz

Le processeur Jazz de Improv Systems[4] est un processeur DSP configurable supportant des instructions VLIW (*Very Long Instruction Word*). Ce processeur possède une grande quantité de modules configurables qui peuvent être ajouté ou non selon les besoins. Puisque ce processeur est de type DSP, la plupart des fonctions optionnelles sont celles que l'on retrouve typiquement dans le domaine du traitement de signal, tels des opérations de multiplication et accumulation, des multiplicateurs et des opérations de décalage de bits. Puisque le Jazz supporte des longues instructions, il est possible de créer soit même sa propre instruction qui regroupe plusieurs

[3] http ://www.stretchinc.com
[4] http ://www.improvsys.com

opérations de base. Tout comme le Xtensa et le ARC Tangent, le Jazz fournit aussi une suite d'outils de développement complète qui s'adapte aux différentes configurations du processeur.

ANNEXE II

APPROCHES POUR MASQUER LA LATENCE

Les techniques présentées à la section 1.2 différentes architectures de processeurs qui permettent d'exploiter différents niveaux de parallélisme et aussi, dans certain cas, de masquer de manière efficace la latence des lectures et écritures. Cependant, dans un SoC, d'autres techniques efficaces peuvent être employées pour masquer ces latences. Par exemple, un processeur VLIW (*Very Long Instruction Word*) travaille sur des instructions très longues qui contiennent plusieurs instructions plus simple. Le compilateur est donc capable d'extraire du ILP sur ce processeur et ainsi faire un travail similaire à ce qui est réalisé dynamiquement sur un processeur superscalaire.

L'autre grande approche pour réduire les latences dans un système est d'utiliser une architecture mémoire spécialisée pour l'application afin de minimiser les temps d'accès. Par exemple, un système multiprocesseur NUMA (Non-Uniform Memory Access) est une solution classique. Dans ce type d'architecture, toute la mémoire est partagée, mais chaque processeur peut accéder à une certaine portion de la mémoire (typiquement un bloc de mémoire local au processeur) beaucoup plus rapidement qu'au reste de la mémoire [91]. Différents protocoles peuvent être utilisés pour échanger des messages entre les processeurs et il est, bien entendu, important de diviser son application pour minimiser le nombre d'accès qu'un processeur doit faire à l'extérieur de sa mémoire locale. Utiliser une mémoire cache pour chaque processeur de la plate-forme est aussi un bon moyen d'augmenter les performances. Dans ce cas, afin de préserver la cohérence des données dans le système où la mémoire est partagée, des mémoires tel une «*snooping cache*» [84] peuvent être utilisées. Cela évite au programmeur d'avoir à se préoccuper de partitionner les données sur chacun des processeurs. Les mémoires caches (et les mémoires distribués) offrent un

bon gain en performance pour des applications typiques. Cependant, le gain n'est pas nécessairement aussi grand pour une application orientée flot de donnée avec beaucoup d'entrées et de sorties de données, comme c'est le cas pour un routeur.

Bien qu'intéressantes et dignes de mentions, ces technique n'ont pas été étudiées pour cette recherche et ne seront donc pas approfondies. Nous notons simplement que ces architectures mémoires peuvent facilement être combinées avec des processeurs supportant plusieurs processus concurrents.

ANNEXE III

OUTILS DE DÉVELOPPEMENT DE HAUT NIVEAU

III.1 Outils de développement commerciaux

Coware N2C et ConvergenceSC. Coware N2C System Designer[1] est une suite d'outils permettant de concevoir un SoC au niveau système en utilisant principalement le langage C (SystemC ou CoWareC) ainsi que des langages HDL (VHDL, Verilog) si nécessaire puisque la co-simulation entre ces différents langages est supportée. CoWare propose un flot de codesign logiciel/matériel semi automatisé où les interfaces entre les blocs logiciels et matériels sont générés par l'outil, cela inclut non seulement les interfaces matérielles, mais aussi le code de bas niveau utilisé par le processeur pour la communication et la synchronisation. CoWare N2C supporte aussi le raffinement d'une implémentation haut niveau vers du RTL grâce à sa technologie nommée Register Transfer C. L'outil offre aussi une interface graphique qui permet de facilement assembler les différentes parties de la plate-forme même si elles sont implémentées à différents niveaux d'abstractions. De son côté, ConvergenceSC permet de simuler une plate-forme décrite en SystemC et d'aller chercher une grande quantité d'informations de profilage et d'analyse. Il offre aussi des outils de déverminage et de vérification adaptés au langage SystemC. De plus, ces outils peuvent être utilisés avec l'environnement de création de processeur embarqué LISATek (voir section 1.1.2).

Proilog Nepsys et Magillem. La compagnie Prosilog[2] offre deux produits qui possèdent plusieurs similitudes avec les outils N2C et ConvergenceSC de Coware.

[1] http ://www.coware.com
[2] http ://www.prosilog.com

Nepsys est un outil entièrement basé sur SystemC qui vise à faciliter le développement de modules à l'aide de ce langage. Nepsys possède un ensemble de fonctionalités pour faciliter les tests et la vérification de modules SystemC. Cet outil offre aussi une bibliothèque de composants standards comme un modèle transactionnel d'un bus AMBA. De plus, Nepsys peut être utilisé pour effectuer une cosimulation entre du VHDL et du SystemC et pour convertir du code SystemC en VHDL ou en Verilog. Magillem, quant à lui, permet de rapidement construire une plate-forme complète en SystemC à partir de modules construits dans Nepsys ou encore réalisés à la main. Magillem offre une interface graphique pour faciliter la connection des différents modules ensemble. Il permet aussi d'automatiquement générer et raffiner les interfaces entre les modules. Magillem supporte plusieurs interfaces standards tels AMBA, OCP et VCI.

Summit Design VisualElite et System Architect. Les outils VisualElite et System Architect de Summit Design[3] offrent des fonctionalités comparables aux outils de Prosilog. VisualElite permet, comme son nom le suggère, de construire graphiquement une plate-forme à partir de modules qui peuvent être décrits en VHDL, C/C++ ou SystemC. De son côté, System Architect est axé sur la vérification et le profilage des différents modules décrits dans un langage de haut niveau.

Cadence VCC. VCC Cierto de Cadence[4] est un autre exemple d'un outil supportant une spécification d'un SoC en C++. Le principe de base de VCC est d'intégrer des blocs déjà existants dans la plate-forme et de les simuler à un niveau fonctionnel. Lorsque les métriques de performances obtenues de ces simulations sont satisfaisantes, VCC supporte le raffinement vers du code RTL ou un RTOS (pour le logiciel). Il offre aussi la possibilité de facilement déplacer des blocs du logiciel au matériel et vice-versa puisque c'est VCC qui s'occupe de la gestion des communications entre les blocs. VCC reste cependant un outil très complexe à utiliser et, par conséquent,

[3]http ://www.summit-design.com
[4]http ://www.cadence.com

ne connaît pas beaucoup de succès.

Synopsys CoCentric Studio. CoCentric System Studio[5] est un autre environnement de développement basé sur SystemC qui permet la spécification et la simulation d'une plate-forme à différents niveaux d'abstraction. Les spécifications du système à concevoir peuvent être entrées graphiquement et il est possible de puiser dans une banque de données comprenant plusieurs algorithmes fréquemment utilisés. CoCentric dispose aussi d'un environnement de simulation et de vérification avancé pour facilement créer des bancs d'essais et aller chercher plusieurs statistiques (le trafique sur les bus par exemple) lors de la simulation. Finalement, il est possible de lier CoCentric à des simulateurs externes tel ModelSim pour co-simuler du SystemC et d'autres langages HDL. La suite d'outils de CoCentric comprend également un outil de synthèse pour le code SystemC. Bien entendu, afin d'être synthétisable, un code SystemC doit se restreindre à un sous-ensemble du langage et se soumettre aux constructions supportées par l'outil de synthèse, comme c'est le cas pour le VHDL.

Seamless CVE et C-Bridge. Seamless CVE [6] de Mentor Graphics est un outil qui permet d'effectuer une co-simulation entre du matériel décrit en code RTL et du logiciel. Le logiciel roule sur un des processeurs supportés par Seamless grâce à un ISS qui possède une interface précise au niveau des broches. Seamless est un outil très utile pour simuler une plate-forme complètement raffinée sous forme de code RTL et presque prête à être réalisée. Une de ses principales forces est qu'il permet d'optimiser la simulation, par exemple en simplifiant la simulation des accès mémoires du processeur. Cependant l'outil s'avère moins utile pour le design au niveau système et l'exploration architecturale. Plus récemment, l'extension C-Bridge qui permet de simuler du code écrit en C/C++ avec le reste de la plate-forme RTL a été lancée. Cette extension permet, à l'aide d'une bibliothèque et d'une interface fournies par Seamless, de communiquer avec le reste de la plate-forme soit en utilisant un bus

[5]http ://www.synopsys.com
[6]http ://www.mentor.com/seamless

prédéfini, soit en utilisant une série de broches personnalisées. Il n'est cependant pas particulièrement facile de réaliser une simulation complexe tout en utilisant l'interface au niveau des broches. Finalement, Seamless offre aussi du support pour le profilage d'une application.

III.2 Outils de développement académiques

Il existe une quantité incroyable d'outils issus du domaine académique visant à faciliter un aspect ou un autre du développement d'un système monopuce et du codesign en général. Nous présentons ci-dessous un petit échantillon de ces outils.

Roses. L'outil Roses [13], développé par le TIMA, s'attarde principalement au raffinement d'une plate-forme définie à haut niveau vers une architecture plus raffinée pouvant être simulée et synthétisée. L'architecture de la plate-forme à raffiner peut être initialement décrite à l'aide d'une version étendue de la bibliothèque SystemC. Par la suite, Roses offre des mécanismes de génération automatique des adaptateurs matériels (afin de communiquer sur un bus AMBA) et des API (Application Programming Interface) logiciels [46].

Ptolemy. Ptolemy [37] est un outil développé à l'université Berkeley de Californie qui a comme objectif de permettre d'expérimenter différentes techniques de design pour les systèmes embarqués. Ptolemy permet de modéliser des systèmes hétérogènes comportant entre autre des éléments analogiques, de la logique électronique et du logiciel. L'emphase est particulièrement mise sur la méthodologie à suivre pour réaliser le système embarqué. Le système peut donc être spécifié en choisissant parmi un grand nombre de modèles de calculs (model of computation) tels les processus de Kahn, une machine à état fini ou un modèle à événements discrets [22]. Pour ces modèles de calculs, une approche dite « orientée acteur » est utilisée ; les différents modules du système s'échangent des messages (possiblement concurrents) à travers des ports et

des canaux de communication (comme c'est le cas avec SystemC). Ptolemy offre aussi une interface graphique évoluée afin de créer et de visualiser le système développé.

MESCAL. Le projet MESCAL (*Modern Embedded Systems : Compilers, Architectures, and Languages*) de l'université de Berkeley [7] a comme objectif de faciliter, à travers le développement d'outils et de méthodologies, le développement de plates-formes programmables et réutilisables pour différentes classes d'applications. Tipi (autrefois nommé Teepee) [51], est un environnement de développement créé dans le cadre du projet MESCAL. À partir d'une description utilisant un langage de description formel, différentes parties de l'architecture peuvent être automatiquement générées. L'outil vise à introduire une approche formelle et structurée pour l'exploration architecturale lors du développement de la plate-forme et se concentre plus particulièrement sur les processeurs ASIP. Le langage Teepee n'est pas sans analogies avec LISA (voir section 1.1.2) puisque plusieurs modèles, ou «vues», sont extraits de la description et utilisés afin de générer certaines composantes du processeur. Le projet MESCAL travaille aussi sur une méthodologie pour implémenter automatiquement une application avec plusieurs processus concurrents sur une plate-forme architecturale [50]. Dans ces travaux, le logiciel de routage Click (décrit à la section 2.2.3) est utilisé. Finalement, notons que le projet MESCAL se base sur Ptolemy pour offrir une interface graphique.

Metropolis. Metropolis[8] [7] est un autre environnement de développement qui est axé sur la spécification et le raffinement de plates-formes réutilisables pour toute une classe d'applications. L'exploration architecturale débute par la description de la plate-forme à l'aide d'un langage formel, le produit de cette étape est le «meta-modèle». Ce modèle permet de décrire la plate-forme à différents niveaux de raffinement et il reste cohérent lors des phases de raffinement successives. Le langage de description formel de Metropolis est lui aussi basé sur la séparation des communica-

[7] http ://www.gigascale.org/mescal
[8] http ://www.gigascale.org/metropolis

tions et des calculs, comme c'est le cas avec MESCAL. De plus, Metropolis offre des outils permettant de simuler la plate-forme, d'effectuer de la vérification et aussi de raffiner le meta-modèle vers une description synthétisable.

SPACE. L'outil SPACE (*SystemC Partitioning of Architectures for the Codesign of Embedded systems*) [14, 9, 71], développé à l'École Polytechnique de Montréal, se penche plus particulièrement sur le problème de partitionnement logiciel/matériel dans un SoC. SPACE offre dans un premier temps une méthodologie pour spécifier une plate-forme à haut niveau en s'appuyant sur la bibliothèque de modélisation SystemC. Par la suite, un deuxième niveau d'abstraction plus détaillé permet de simuler la plate-forme avec ses composantes logicielles s'exécutant sur un ISS et le reste de la plate-forme étant simulé avec SystemC. L'avantage de ce deuxième niveau est qu'il est associé à une méthodologie et à une infrastructure permettant de rapidement et facilement simuler plusieurs scénarios de partitionnement logiciel/matériel. Cela est possible grâce à l'utilisation d'un RTOS ayant la même API (application programming interface) que SystemC, les modules peuvent donc être directement déplacés du monde matériel au monde logiciel.

ANNEXE IV

DIFFÉRENTS COMPOSANTS DE STEPNP

IV.1 La bibliothèque SystemC

Un bon nombre de raisons font de SystemC un choix intéressant pour la description des modules de StepNP (ou de n'importe quel autre environnement de développement au niveau système). Nous notons entre autre que SystemC :

- permet l'utilisation des techniques de programmation orientée objet pour la spécification fonctionnelle ;
- offre une grande variété de niveau d'abstraction pour la description du circuit ;
- permet le raffinement successif d'une spécification ;
- est en voie de devenir un standard dans l'industrie et dans les universités pour la spécification fonctionnelle ;
- est développé publiquement par le OSCI (*Open SystemC Initiative*).

SystemC permet de décrire un système numérique à plusieurs niveaux d'abstraction. Au plus bas niveau, que l'on peut nommer RTL (de l'anglais Register Transfer Level), SystemC ressemble beaucoup et exploite les même concepts que des langages tels VHDL et Verilog. Le design est divisé en modules afin de partitionner un design complexe en sous-blocs et contrôler la complexité. Chaque module, contient généralement des ports d'entrées et des ports de sorties, des variables locales, des signaux ainsi que des processus. Un module peut aussi contenir des modules hiérarchiques et des signaux internes qui sont utilisés pour communiquer avec ces derniers. Chaque module contient un certain nombre de processus qui permettent de décrire des algorithmes d'exécutions sensibles à des événements ou des signaux externes ou internes. Ces processus sont très similaires au processus du VHDL et apporte les mécanismes nécessaires pour une simulation en parallèle. Un système numérique décrit à ce ni-

veau d'abstraction peut être synthétisé à l'aide d'outils commerciaux en autant qu'il respecte certaines contraintes (par exemple au niveau du type des variables employées). La figure IV.1 illustre un système décrit à l'aide de modules, de ports et de signaux.

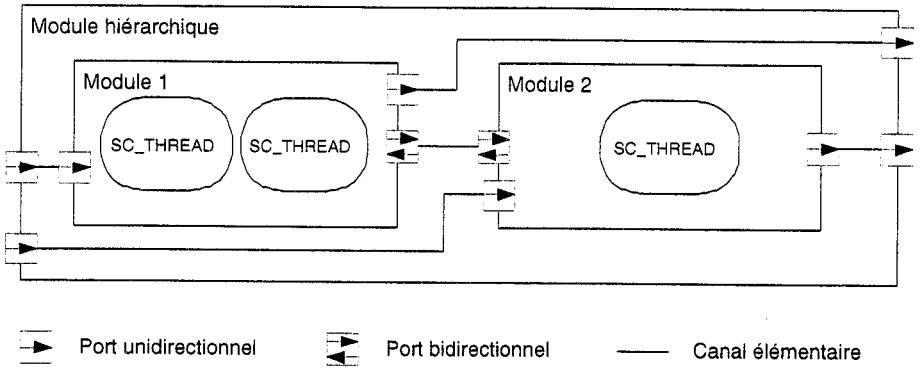


FIGURE IV.1 Module SystemC hiérarchique utilisant des signaux

Cependant, lorsque nous effectuons de l'exploration architecturale il est préférable d'utiliser un plus haut niveau d'abstraction afin de décrire le système. SystemC offre la possibilité de créer une description purement fonctionnelle à un niveau d'abstraction nommé UTF et TF (respectivement Untimed Functional et Timed Functional). La principale différence entre le niveau UTF et le niveau TF est que le second possède des délais qui permettent d'estimer la latence de l'exécution. Ces deux niveaux ne contiennent pas de notion d'horloge ou de cycle et les algorithmes peuvent être décrits en C++ en utilisant toutes les fonctionnalités de ce langage orienté objet. Cela permet de décrire des modules complexes beaucoup plus rapidement qu'avec les contraintes du niveau RTL, mais la possibilité de synthétiser automatiquement est sacrifiée. Ce qui distingue le plus les niveaux UTF et TF vis-à-vis une description RTL est cependant le fait qu'ils emploient une communication au niveau transactionnelle nommée TLM (*Transaction Level Modeling*). Avec des communications TLM, les modules communiquent entre eux via un canal abstrait qui modélise uniquement les échanges de données. Dans un modèle TLM décrit en SystemC, un canal abstrait implémente une interface et différents modules possèdent un port qui utilise cette

interface pour communiquer. Une interface est simplement un ensemble de fonctions décrites en C++ qui permettent d'envoyer ou recevoir des données selon un certain format prédéterminé. Ce type de communication est à la base de tous les échanges dans StepNP et est illustré à la figure IV.2.

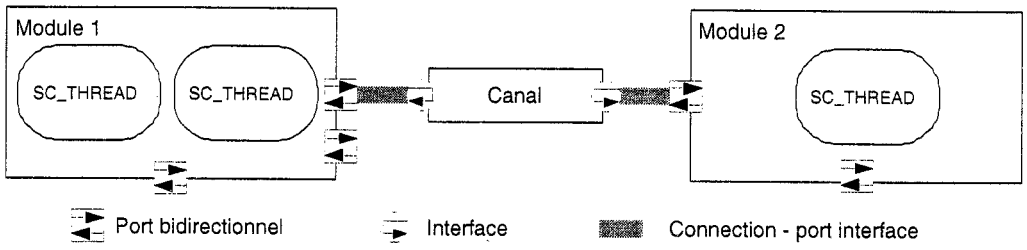


FIGURE IV.2 Modules SystemC utilisant des ports et des interfaces

À la base, avec des canaux de communication TLM, la synchronisation se fait uniquement sur les données qui sont échangées puisqu'il n'y a aucune notion d'horloge. Il est cependant possible d'utiliser des composantes à différents niveaux d'abstractions ensemble, par exemple un processeur simulé sur un ISS (précis au cycle d'horloge) peut communiquer avec une mémoire UTF à travers le canal simplement en utilisant des fonctions de lecture et d'écriture. De plus, il est possible de raffiner le canal TLM, tout en conservant son interface, afin qu'il devienne un modèle lui aussi précis au niveau des cycles (appelé BCA ou Bus Cycle Accurate). Dans StepNP, la majorité des éléments existants sont soit UTF/TF (par exemple une mémoire ou un canal) ou BCA (un processeur ou un canal plus raffiné). Un même élément peut être soit UTF ou TF lorsqu'il possède un paramètre configurable qui permet de lui attribuer une latence ou non.

En résumé, SystemC ajoute plusieurs fonctionnalités au langage C++ qui sont typiques d'un langage HDL dont :

- des types de données propres aux descriptions matérielles (sc_bit, sc_bit_vector, sc_logic, sc_int, ...) ;
- des modules, l'élément structurel de base de SystemC (sc_module) ;
- des processus qui permettent de décrire des algorithmes concurrents (sc_thread,

sc_cthread, sc_method) ;

- des signaux qui permettent la communication inter-processus ou inter-module (sc_signal) ;
- des éléments de synchronisation qui permettent de suspendre l'exécution des processus perpétuels (wait ou sc_event).

De plus, SystemC offre certaines constructions qui offrent un plus grand niveau d'abstraction que ce que l'on retrouve typiquement dans un langage HDL comme Verilog ou VHDL :

- des ports d'entrées et de sorties qui peuvent se connecter soit à un signal (sc_in, sc_out) ou à une interface d'un canal (sc_port) ;
- des canaux abstraits et des interfaces qui y sont associées (sc_canal et sc_interface) ;
- des objets prédéfinis qui offrent des fonctionnalités plus avancées telle une file (sc_fifo).

Outre l'avantage évident qu'il est plus facile de concevoir une architecture d'un SoC avec un plus haut niveau d'abstraction qui masque plusieurs éléments de complexité, la simulation au niveau TLM offre aussi des vitesses de simulation beaucoup plus rapides que celles obtenues avec un modèle raffiné comme du code RTL. En effet, échanger un ensemble de données à l'aide d'un simple appel de fonction C++ demande beaucoup moins de temps qu'au niveau RTL où chaque signal doit être simulé. Pour une plate-forme complexe qui intègre des éléments logiciels et matériels, la simulation peut être plusieurs ordres de grandeurs plus rapide. Finalement un autre avantage de SystemC est qu'en plus de permettre une description à plusieurs niveaux d'abstraction et d'être bien adapté pour modéliser la concurrence dans une architecture, il n'impose pas de restrictions sur le modèle de calcul employé (aussi appelé MoC ou model of computation, c'est-à-dire le modèle mathématique utilisé pour décrire la concurrence et le fonctionnement du système). Par exemple, il n'est pas nécessaire de se limiter à un certain type de machine à état imposé par l'outil. De plus, avec une approche de conception basée sur les communications TLM, différents modules utilisant différents modèles de calculs peuvent facilement communiquer entre eux.

IV.2 Plate-forme de développement logiciel : Click

Disposer d'une plate-forme matérielle, qu'elle soit décrite à haut niveau d'abstraction ou non, est très utile, voir nécessaire, pour pouvoir développer efficacement le logiciel du système embarqué. Cependant cela n'est pas suffisant pour être capable de développer rapidement le logiciel et pour faciliter la réutilisation du logiciel à travers différents projets. Pour ce faire, il est nécessaire d'offrir une interface de programmation (un API) que l'on peut aussi appeler un modèle de programmation. Ce modèle permet au programmeur de voir uniquement une abstraction de la plate-forme matérielle ce qui facilite énormément le développement du logiciel. Si l'on revient aux NPU décrits à la section 1.3, il apparaît que la plupart de ces plates-formes offrent des outils de conception logicielle. Au minimum, un compilateur C ou C++ est disponible, mais dans bien des cas le programmeur doit encore utiliser du langage assembleur pour tirer avantage des particularités d'un NPU donné. Cela réduit grandement la productivité du programmeur puisqu'il est forcé de bien connaître l'architecture matérielle et aussi de coder en langage machine.

Une première approche pour palier à ce problème est d'utiliser un système d'exploitation temps réel (RTOS) qui permet de distribuer les tâches à accomplir et offre aussi des méthodes d'accès aux différents modules d'entrées / sorties et aux différents périphériques de la plate-forme. Le RTOS agit alors comme couche logicielle qui permet d'abstraire les détails du matériel. Cette approche très utile est mise de l'avant dans plusieurs projets dont SPACE [14]. Cependant, dans une architecture multiprocesseur, il devient plus difficile pour le RTOS de distribuer efficacement les tâches logicielles dans la plate-forme. Une extension de StepNP, nommée DSOC (*Distributed system-on-chip*) se penche sur ce problème. Avec DSOC, chaque processeur dispose d'un très mince RTOS qui utilise des accélérateurs matériels afin d'offrir la possibilité de créer des processus concurrents et d'échanger des messages très rapidement entre les processus. DSOC, décrit dans [62], propose aussi une méthodologie pour facile-

ment et efficacement distribuer un ensemble de tâches sur un groupe de processeurs et de coprocesseurs. Bien que très intéressante et prometteuse, cette approche n'a pas été utilisée dans le cadre de ce mémoire et ne sera donc pas approfondie.

Une autre possibilité pour offrir un modèle de programmation est d'utiliser un environnement de développement qui permet de rapidement créer une application (ne reposant pas nécessairement sur un RTOS) à l'intérieur d'une certaine classe d'applications tels les processeurs réseau. Cet environnement de développement doit être portable et de haut niveau puisqu'il sera appelé à être exécuté sur différents processeurs. Pour ces raisons, StepNP utilise Click [45], un environnement originalement développé au MIT, qui permet de rapidement créer différentes applications réseau. Click peut être décrit comme un routeur modulaire puisqu'il est composé d'un ensemble d'éléments réseau qui peuvent être connectés ensembles afin de créer le type de routeur désiré. Les éléments Click sont programmés en C++ et offrent toutes les fonctionnalités que l'on retrouve dans un routeur dont le filtrage et la modification de paquets, la logique de contrôle et les communications avec les interfaces. Pour l'instant, Click fournit plus de 250 éléments permettant de manipuler les différents paquets et il est facile d'ajouter soit même des nouveaux éléments. Click fournit aussi un langage de configuration simple qui permet de spécifier quels éléments sont utilisés dans un routeur donné et comment ils sont connectés entre eux. À l'exécution, ce fichier est analysé afin de produire un routeur au comportement désiré. Il s'agit donc d'un environnement très flexible qui permet de rapidement générer le code d'un processeur réseau. La figure 2.1(b) illustre une configuration de Click qui contient, entre autre, un élément *Classifier* qui rejette les paquets qui ne sont pas des paquets IP et un élément *Strip* qui enlève l'entête Ethernet du paquet. Dans StepNP, certains éléments de Click ont été modifiés ou ajoutés afin de permettre une communication avec l'extérieur de la plate-forme au travers d'un lien SIDL (décrit à la section IV.3.2) ou encore avec le reste de la plate-forme au travers d'un canal SOCP (décrit à la section 2.3.4).

Bien entendu, la flexibilité offerte par Click implique un coût au niveau des performances vis-à-vis un logiciel codé à la main avec une bonne connaissance de l'architecture du SoC. Cependant, à l'étape de l'exploration architecturale, les différentes architectures étudiées doivent seulement être profilées en utilisant une application réseau représentative. Afin d'écourter la phase d'exploration, il est alors très utile d'avoir rapidement une application qui, bien qu'elle ne soit pas dans sa version finale, permet de discriminer entre les différentes solutions. Click s'acquitte très bien de cette tâche. De plus, comme il sera démontré au chapitre 3, en utilisant un jeu d'instructions spécialisé, il est possible d'utiliser Click tout en optimisant les boucles critiques. Ceci permet d'obtenir de bonnes performances tout en conservant la flexibilité. Un NPU capable d'atteindre des performances intéressantes a aussi été développé en utilisant Click par l'équipe de StepNP [63]. Ce NPU tire profit de la méthodologie d'objets distribués DSOC ainsi que de processeurs HMT.

Il est donc possible d'utiliser Click, non seulement comme outil de prototypage rapide d'une application réseau, mais aussi comme interface haut niveau qui permet de masquer certains détails de la plate-forme matérielle tout en obtenant les performances désirées. De plus, le code ainsi développé reste portable d'une plate-forme à une autre en autant que Click soit adapté à ces différentes plates-formes. Une telle utilisation de Click a aussi été étudiée dans le cadre du projet MESCAL avec le Intel IXP1200 comme plate-forme [77].

IV.3 Outils de contrôle, d'analyse et de vérification

StepNP offre un cadre flexible et puissant pour créer des architectures de NPU (ou d'une autre classe d'application), ainsi que pour créer le code logiciel et pour simuler la plate-forme. Cependant, sans outils de contrôle, de profilage, d'analyse et de vérification permettant de rapidement tirer des conclusions de la simulation, l'avantage d'un environnement comme StepNP se trouve grandement réduit. C'est

pourquoi StepNP offre différentes applications utiles lors de la simulation dont un outil de contrôle, une interface pour communiquer avec les modules en cours de simulation et un environnement graphique pour afficher des traces de simulations.

IV.3.1 Outil de contrôle et de génération de plate-forme : SocGen

Afin de créer une plate-forme complète et simulable dans StepNP, l'usager peut utiliser un fichier source C++ qui inclut la fonction principale de SystemC (`sc_main`) et dans laquelle tous les composants sont déclarés puis connectés les uns aux autres. Bien que simple, cette étape peut devenir longue et fastidieuse surtout lorsque nous désirons effectuer plusieurs simulations en enlevant et en ajoutant des composants ou encore en modifiant les paramètres configurables d'un ou plusieurs des composants. C'est pourquoi un générateur de SoC, nommé SocGen, est inclu dans StepNP. Cet outil permet de créer dynamiquement une plate-forme à l'aide d'un script en langage Python. SocGen permet, dans un premier temps, de facilement ajouter des composants dans une plate-forme et d'initialiser leurs paramètres (telle la latence d'exécution). Dans un second temps, SocGen permet aussi de contrôler dynamiquement des paramètres de la simulation ce qui peut s'avérer un avantage non négligeable. SocGen permet aussi d'activer dynamiquement des sondes à l'intérieur de la plate-forme qui recueillent des informations sur l'état de certains modules ou signaux. Afin de supporter ces nouvelles fonctionnalités, certains éléments ont dû être ajoutés à SystemC, dont un serveur de noms qui permet à SocGen de trouver un composant à partir de son nom ainsi que tout le nécessaire pour changer et décharger dynamiquement un module SystemC. Encore une fois, des classes de base qui permettent de facilement créer un nouveau module matériel supportant SocGen sont fournies dans StepNP.

IV.3.2 Interface de communication : SIDL

SIDL (*SystemC Interface Definition Langage*) est, comme son nom l'indique, un langage permettant de définir une interface entre une simulation SystemC et un programme externe qui peut être écrit dans un langage différent du C++. Le concept de base de SIDL est similaire à d'autres langages IDL bien connus tel CORBA, Java RMI ou encore Microsoft DCOM. La différence principale entre SIDL et ces langages est d'abord que ce dernier est basé sur SystemC, mais surtout qu'il est beaucoup plus réduit dans ses fonctionnalités et donc plus rapide et léger à utiliser [60].

Le code présenté à la figure IV.3 illustre une interface qui permet de lire et de modifier le compteur de programmes d'un processeur en cours d'exécution dans StepNP. Cette définition se fait en utilisant la syntaxe du C++ et est très semblable à la définition d'une interface SystemC (`sc_interface`).

```
class processorIF {
    public :
        virtual int readPC() = 0;
        virtual int setPC(int) = 0;
}
```

FIGURE IV.3 Exemple d'une interface SIDL

À partir de cette définition, le compilateur SIDL génère tout le code nécessaire pour permettre une communication entre un client et un serveur qui utilisent cette interface. Dans le cas de notre interface, le processeur serait le client alors qu'un logiciel externe qui peut être écrit en C++, Java ou Python, serait le serveur qui inspecte ou modifie l'état du processeur en cours d'exécution. Le client et le serveur SIDL communiquent ensemble en utilisant le mécanisme des RPC (*remote procedure call*) fournit par le système d'exploitation de la machine hôte à la simulation. L'utilisation de SIDL permet donc de :

- intégrer la simulation SystemC avec d'autres éléments simulés à l'extérieur de

SystemC ;

- modéliser des interfaces d'entrées/sorties externes à la plate-forme simulée ;
- distribuer la simulation sur plusieurs machines ;
- connecter des outils d'introspection et de visualisation à la simulation (ceci est utilisé par SocMon) ;
- contrôler la simulation à l'aide de scripts externes à SystemC (ceci est utilisé par SocGen).

Notons finalement que la méthodologie DSOC dont nous avons brièvement discuté à la section 2.2.3 utilise aussi des interfaces SIDL afin d'assurer les communications entre les différents processus dans la plate-forme. De cette manière, la communication reste transparente peu importe si les modules sont implémentés en matériel ou s'ils roulent sur un processeur.

IV.3.3 Environnement de visualisation : SocMon

L'environnement de visualisation SocMon (*SoC Monitor*) est illustré à la figure 2.1(c). Il s'agit d'une application Java qui permet d'afficher des informations adaptées à chaque plate-forme. En effet, à l'aide d'interfaces d'introspection SIDL, l'usager peut définir le type d'informations qui seront automatiquement extraits et affichés par SocMon en cours de simulation et ainsi afficher uniquement les informations voulues. SocMon est conçu pour offrir plusieurs types de vues sur la plate-forme et chaque type de vue est spécialisé pour afficher différentes informations. Les types d'affichage sont :

- une vue temporelle qui permet d'obtenir la trace de certaines variables ;
- une vue logicielle qui permet de voir les registres d'un processeur supportant une interface SIDL compatible ;
- une vue applicative du code source qu'un processeur exécute à un instant donné.

ANNEXE V

LE PROTOCOLE IPSEC ET LE CHIFFREMENT DES

V.1 Description de IPSec

Le protocole IPSec est décrit en détail dans la norme RFC2401 [42]. L'objectif de ce protocole est d'offrir des services d'authentification et de confidentialité (chiffrement) afin de pouvoir échanger des données en tout sécurité sur un réseau publique. Le protocole d'authentification est décrit dans la norme RFC2402 [40]. Les deux principales tâches de ce protocole sont de permettre au récepteur d'authentifier avec certitude la source des données et d'assurer l'intégrité des données. L'authenticité des données est assurée grâce à un mécanisme permettant de détecter toutes modifications apportées par un tiers parti qui aurait intercepté la communication. Le mécanisme de chiffrement des données est quant à lui décrit par le protocole EPS (*Encapsulating Security Payload*), défini dans la norme RFC2406 [41]. Le protocole EPS définit deux modes de fonctionnement : le mode transport et le mode tunnel. Dans le mode transport, seule les données sont encryptées. L'en-tête IP, qui permet au différents routeurs d'acheminer le paquet a destination, reste lisible. En mode tunnel, tout le paquet est encrypté et une nouvelle en-tête est ajoutée. Cette nouvelle en-tête permet à un routeur donnée d'envoyer le paquet à sa prochaine destination dans le réseau tout en masquant l'adresse d'origine et de destination finale ce qui assure une confidentialité limitée. Le chiffrement des données en tant que tel peut être réalisée à l'aide de plusieurs algorithmes dont DES (*Data Encryption Standard*), Triple-DES, RC5, IDEA (*International Data Encryption Algorithm*) et *Blowfish*

V.2 Algorithme DES

L'algorithme de chiffrement DES est décrit en détail par l'organisme américain *National Institute of Standards and Technology* (NIST). L'objectif ici est de donnée suffisamment de détails sur cet algorithme pour bien illustré pourquoi il est difficile de le réaliser en logiciel et pourquoi les instructions TIE sont tout indiquées pour l'optimiser. Le lecteur désireux de connaître tous les menus détails peut se référer aux normes FIPS46-2 et 46-3 [54].

V.2.1 Gestion des clés

Pour encrypter un bloc de 64 bits de données, DES utilise une clé de 64 bits. La clé doit être générée aléatoirement au début de la communication et échangée (préférablement de manière confidentielle) entre les deux machines. DES fonctionne avec une clé symétrique, c'est-à-dire que l'émetteur et le récepteur possèdent tous les deux la clé et qu'elle peut être utilisée pour encrypter et pour décrypter les données. Cela diffère des mécanisme, plus sécuritaire et plus lourd a gérer, ou chaque machine possède une clé publique et une clé privée. DES utilise en réalité une clé de 56 bits pour le chiffrement, puisque 8 bits dans la clé sont des bits de parité qui servent uniquement a détecter s'il y a eu une erreur lors de la transmission de la clé. L'algorithme DES consiste principalement à appliquer 16 fois la même transformation sur le paquet à l'aide de 16 sous-clés différentes. La première étape à réaliser lors de la réception de la clé est donc de générer les 16 sous-clés à partir de cette dernière.

La figure V.1 montre les principales étapes nécessaires pour générer les 16 sous clés. Tout d'abord, une permutation est effectuée sur tous les bits de la clé, les 8 bits de parité sont ignorés par cette permutation qui produit donc un mot de 56 bits. Déplacer tous les bits d'un mot selon un patron irrégulier est une opération complexe à réaliser en logiciel, mais triviale avec du matériel (et donc avec une instruction TIE).

Par la suite le mot de 56 bits est séparé en deux mots de 28 bits (nommées C et D). Les bits de ces deux mots subissent une rotation de 1 ou 2 bits vers la gauche. Ils sont ensuite recombinés ensemble pour subir une permutation qui va produite la première sous-clé de 48 bits. Cette procédure est répétée 16 fois, tout en changeant la taille de la rotation à chaque nouvelle itération, afin de produire les 16 sous-clés.

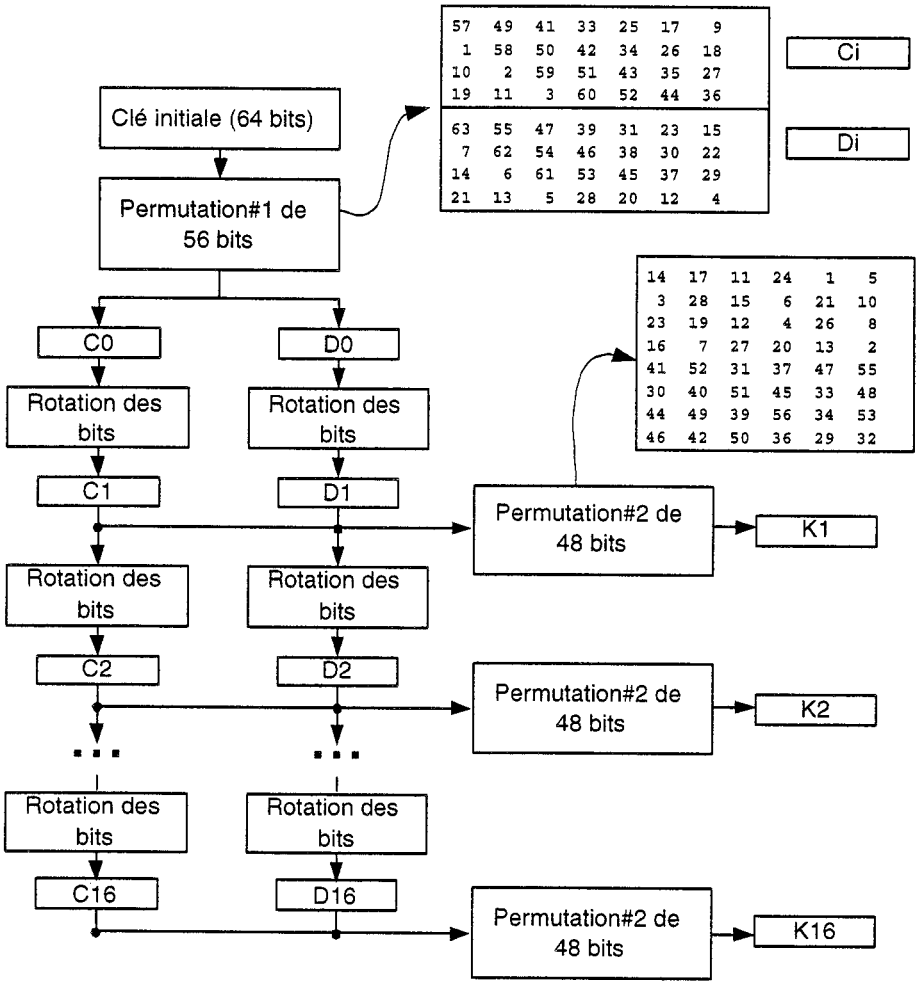


FIGURE V.1 Étapes pour la génération des sous-clés

V.2.2 Transformation des données

Les 16 sous-clés peuvent être utilisées pour encrypter plusieurs blocs de 64 bits de données. La clé initialement échangée entre les deux machines est valide pour une

session de communication entière. Puisque les même 16 mots de 48 bits vont être accédés très fréquemment, notre instruction TIE les sauvegardes dans des registres dédiés. Le chiffrement des données se fait en répétant 16 fois la même fonction de chiffrement $f(R_i, K_i)$ avec une clé différente à chaque itération. La figure V.2 montre les principales étapes à suivre. Tout d'abord, une permutation initiale est effectuée sur les données, avec cette permutation, le bit 58 devient le premier bit du mot L et le bit 57 devient le premier bit du mot R (et ainsi de suite). À chaque itération, la fonction d'encryptions $f(R_i, K_i)$ est appliquée sur le bloc R de 32 bits et la clé K de 48 bits. Par la suite un «ou exclusif» est réalisé et les deux mots R et L sont inversés. La formule générale est donc :

$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus f(R_i, K_i)$$

À la fin du traitement, l'inverse de la permutation initiale est appliqué sur les données. Cela permet au récepteur de déchiffrer les données en appliquant exactement le même algorithme, seul les blocs L et R doivent être intervertis.

V.2.3 Fonction de chiffrement

La fonction de chiffrement $f(R_i, K_i)$ produit un mot de 32 bits à partir d'une clé de 48 bits et d'un bloc de données de 32 bit. Pour ce faire, des permutations sont effectuées ainsi que des accès dans des tables de constantes. Tel que présenté dans le code source TIE à la section suivante, 8 mots de 6 bits sont générés et ces mots sont utilisés pour aller chercher des mots de 4 bits dans des tables prédéfinies. Encore une fois, il s'agit là d'opérations qui sont difficiles à réaliser avec un jeu d'instructions conventionnel.

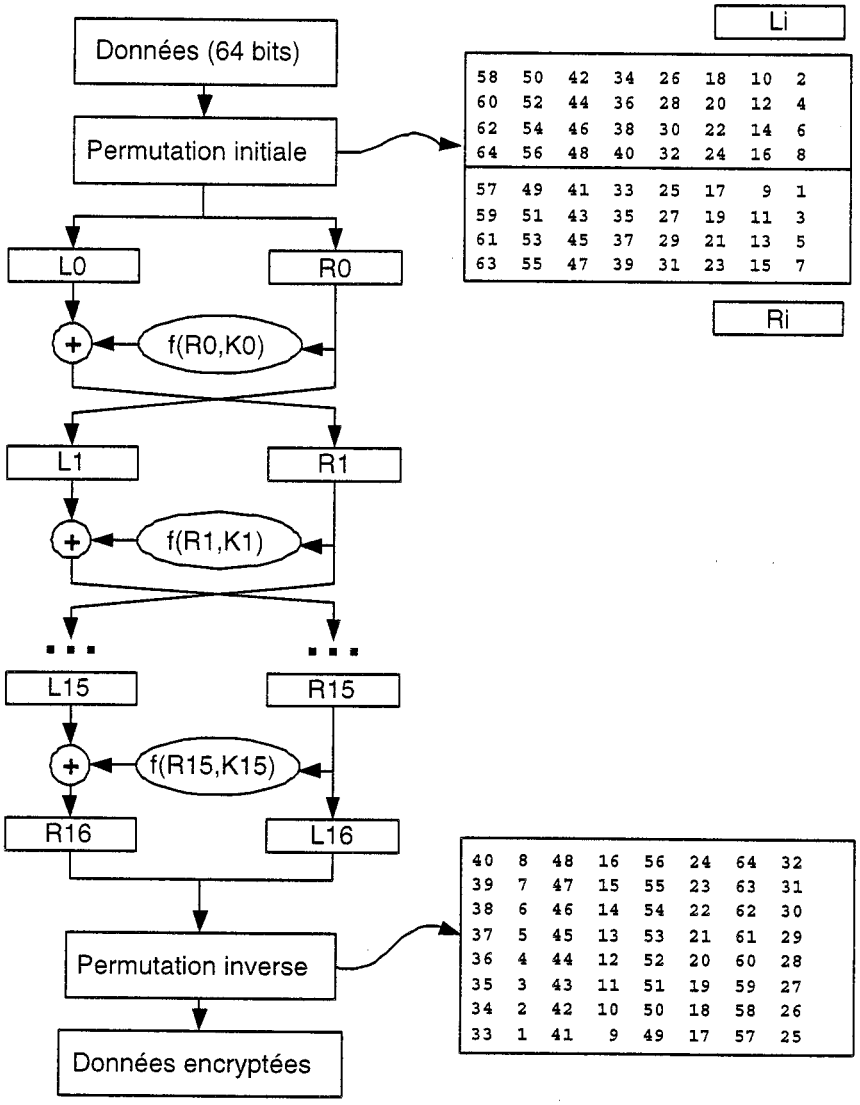


FIGURE V.2 Principales étapes du chiffrement DES

V.3 Code source

V.3.1 Instructions TIE

```
// Date Encryption Standard (DES) utilise dans IPSEC (IP Security)
// FIPS 46-3: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
// Note : Dans le document original, la numérotation des bits dans un mot
//        va de 1 à 64 au lieu de 63 à 0. Les tables dans ces instructions
//        TIE ont donc été inversées.

opcode DES_FCT op2=4'd6 CUST0
opcode DES_KEYGEN op2=4'd7 CUST0
opcode DES_IPa op2=4'd8 CUST0
```

```
opcode DES_IPb op2=4'd9 CUST0
opcode DES_REVIPa op2=4'd10 CUST0
opcode DES_REVIPb op2=4'd11 CUST0

state KEYSC  768   // Contient les 16 sous-clés du key schedule

// Les 4 bits du champs standard "t" sont utilisées ici comme valeur immédiate
// pour sélectionner laquelle des 16 sous-clés sera utilisée. Aucune section
// "operand" n'est nécessaire pour encoder où décoder ce champ.
field keynum {t}
// Définition des classes d'instructions
iclass ipsec_a {DES_FCT} {out arr, in ars, in keynum} {in KEYSC}
iclass ipsec_b {DES_KEYGEN} {in ars, in art} {out KEYSC}
iclass ipsec_c {DES_IPa, DES_IPb, DES_REVIPa, DES_REVIPb}
               {out arr, in ars, in art}

// Les huit tables utilisées pour les transformations Sx
table s1 4 64 {
14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}
table s2 4 64 {
15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}
table s3 4 64 {
10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}
table s4 4 64 {
7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}
table s5 4 64 {
2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}
table s6 4 64 {
12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4 ,10, 1, 13, 11, 6,
4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}
table s7 4 64 {
4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}
table s8 4 64 {
13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}


// ***************************************************************************
// Fonction de chiffrement principale, utilisée pour caclculer f(R,K)
// R est dans le register ars
// K est dans le registre KEYx
// ***************************************************************************
reference DES_FCT {
   // Multiplexeur pour sélectionner la bonne clé.
   wire [47:0] K = TIEmux(keynum, KEYSC[47:0], KEYSC[95:48], KEYSC[143:96],
                          KEYSC[191:144], KEYSC[239:192], KEYSC[287:240],
                          KEYSC[335:288], KEYSC[383:336], KEYSC[431:384],
                          KEYSC[479:432], KEYSC[527:480], KEYSC[575:528],
                          KEYSC[623:576], KEYSC[671:624], KEYSC[719:672],
                          KEYSC[767:720]);

   // Expansion et permutation de R
   wire [31:0] R = ars;
   wire [47:0] E = {
      R[ 0], R[31], R[30], R[29], R[28], R[27], R[28], R[27],
      R[26], R[25], R[24], R[23], R[24], R[23], R[22], R[21],
      R[20], R[19], R[20], R[19], R[18], R[17], R[16], R[15],
      R[16], R[15], R[14], R[13], R[12], R[11], R[12], R[11],
      R[10], R[ 9], R[ 8], R[ 7], R[ 8], R[ 7], R[ 6], R[ 5],
      R[ 4], R[ 3], R[ 4], R[ 3], R[ 2], R[ 1], R[ 0], R[31]};
```

```
    // Ou-exclusif entre R et la clé
    wire [47:0] EK = E ^ K;

    // Calcul des indx pour les tables Sx
    wire [5:0] s1i = {EK[47],EK[42],EK[46:43]};
    wire [5:0] s2i = {EK[41],EK[36],EK[40:37]};
    wire [5:0] s3i = {EK[35],EK[30],EK[34:31]};
    wire [5:0] s4i = {EK[29],EK[24],EK[28:25]};
    wire [5:0] s5i = {EK[23],EK[18],EK[22:19]};
    wire [5:0] s6i = {EK[17],EK[12],EK[16:13]};
    wire [5:0] s7i = {EK[11],EK[6],EK[10:7]};
    wire [5:0] s8i = {EK[5],EK[0],EK[4:1]};

    // Accès aux tables Sx pour générer le nouveau mot
    wire [31:0] P = {s1[s1i],s2[s2i],s3[s3i],s4[s4i],
                     s5[s5i],s6[s6i],s7[s7i],s8[s8i]};

    // Permutation finale
    assign arr = {
       P[16], P[25], P[12], P[11], P[ 3], P[20], P[ 4], P[15],
       P[31], P[17], P[ 9], P[ 6], P[27], P[14], P[ 1], P[22],
       P[30], P[24], P[ 8], P[18], P[ 0], P[ 5], P[29], P[23],
       P[13], P[19], P[ 2], P[26], P[10], P[21], P[28], P[ 7]};
}

// ****************************************************************************
// Permutation finale des données
// Ici deux fonctions sont utilisées pour travailler sur 32 bits à la fois.
// Il serait aussi possible de se créer des registres personnalisés de 64 bits.
// ****************************************************************************
reference DES_IPa { // Retourne les 32 MSB
    wire [63:0] D = {ars[31:0],art[31:0]};
    assign arr = {
      D[ 6], D[14], D[22], D[30], D[38], D[46], D[54], D[62],
      D[ 4], D[12], D[20], D[28], D[36], D[44], D[52], D[60],
      D[ 2], D[10], D[18], D[26], D[34], D[42], D[50], D[58],
      D[ 0], D[ 8], D[16], D[24], D[32], D[40], D[48], D[56]};
}
reference DES_IPb { // Retourne les 32 LSB
    wire [63:0] D = {ars[31:0],art[31:0]};
    assign arr = {
      D[ 7], D[15], D[23], D[31], D[39], D[47], D[55], D[63],
      D[ 5], D[13], D[21], D[29], D[37], D[45], D[53], D[61],
      D[ 3], D[11], D[19], D[27], D[35], D[43], D[51], D[59],
      D[ 1], D[ 9], D[17], D[25], D[33], D[41], D[49], D[57]};
}

// ****************************************************************************
// Permutation initiale inversée (utilisée comme permutation finale)
// ****************************************************************************
reference DES_REVIPa { // Retourne les 32 MSB
    wire [63:0] D = {ars[31:0],art[31:0]};
    assign arr = {
      D[24], D[56], D[16], D[48], D[ 8], D[40], D[ 0], D[32],
      D[25], D[57], D[17], D[49], D[ 9], D[41], D[ 1], D[33],
      D[26], D[58], D[18], D[50], D[10], D[42], D[ 2], D[34],
      D[27], D[59], D[19], D[51], D[11], D[43], D[ 3], D[35]};

}
reference DES_REVIPb { // Retourne les 32 LSB
    wire [63:0] D = {ars[31:0],art[31:0]};
    assign arr = {
      D[28], D[60], D[20], D[52], D[12], D[44], D[ 4], D[36],
      D[29], D[61], D[21], D[53], D[13], D[45], D[ 5], D[37],
      D[30], D[62], D[22], D[54], D[14], D[46], D[ 6], D[38],
      D[31], D[63], D[23], D[55], D[15], D[47], D[ 7], D[39]};
}

// ****************************************************************************
// Calcul des 16 sous-clés. Elles sont sauvegardés dans un registre dédié
// puisque les même clés sont habituellement utilisées pour un grand nombre de
// donnée
// ****************************************************************************
reference DES_KEYGEN {
    wire [63:0] kin = {ars, art};

    // Permutation initiale, élimine les 8 bits de parité(0,8,16,24,32,40,48,56)
    wire [55:0] pc1 = {
       kin[ 7], kin[15], kin[23], kin[31], kin[39], kin[47], kin[55],
       kin[63], kin[ 6], kin[14], kin[22], kin[30], kin[38], kin[46],
       kin[54], kin[62], kin[ 5], kin[13], kin[21], kin[29], kin[37],
       kin[45], kin[53], kin[61], kin[ 4], kin[12], kin[20], kin[28],
       kin[ 1], kin[ 9], kin[17], kin[25], kin[33], kin[41], kin[49],
```

```
   kin[57], kin[ 2], kin[10], kin[18], kin[26], kin[34], kin[42],
   kin[50], kin[58], kin[ 3], kin[11], kin[19], kin[27], kin[35],
   kin[43], kin[51], kin[59], kin[36], kin[44], kin[52], kin[60]};

// Rotation des bits dans les mots de 28 bits. Les 16 rotations sont de
// 1,2,4,6,8,10,12,14,15,17,19,21,23,25,27 et 28 bits
wire [55:0] k1 =  {pc1[54:28], pc1[55],    pc1[26:0], pc1[27]   };
wire [55:0] k2 =  {pc1[53:28], pc1[55:54], pc1[25:0], pc1[27:26]};
wire [55:0] k3 =  {pc1[51:28], pc1[55:52], pc1[23:0], pc1[27:24]};
wire [55:0] k4 =  {pc1[49:28], pc1[55:50], pc1[21:0], pc1[27:22]};
wire [55:0] k5 =  {pc1[47:28], pc1[55:48], pc1[19:0], pc1[27:20]};
wire [55:0] k6 =  {pc1[45:28], pc1[55:46], pc1[17:0], pc1[27:18]};
wire [55:0] k7 =  {pc1[43:28], pc1[55:44], pc1[15:0], pc1[27:16]};
wire [55:0] k8 =  {pc1[41:28], pc1[55:42], pc1[13:0], pc1[27:14]};
wire [55:0] k9 =  {pc1[40:28], pc1[55:41], pc1[12:0], pc1[27:13]};
wire [55:0] k10 = {pc1[38:28], pc1[55:39], pc1[10:0], pc1[27:11]};
wire [55:0] k11 = {pc1[36:28], pc1[55:37], pc1[8:0],  pc1[27:9] };
wire [55:0] k12 = {pc1[34:28], pc1[55:35], pc1[6:0],  pc1[27:7] };
wire [55:0] k13 = {pc1[32:28], pc1[55:33], pc1[4:0],  pc1[27:5] };
wire [55:0] k14 = {pc1[30:28], pc1[55:31], pc1[2:0],  pc1[27:3] };
wire [55:0] k15 = {pc1[28],    pc1[55:29], pc1[0],    pc1[27:1] };
wire [55:0] k16 = pc1;  // Rotation de 28 bits = aucune rotation

wire [47:0] kout1, kout2, kout3, kout4, kout5, kout6, kout7, kout8,
            kout9, kout10, kout11, kout12, kout13, kout14, kout15, kout16;

// Les 16 sous-clés sont obtenues après une permutation finale
assign kout1 = {
  k1[42], k1[39], k1[45], k1[32], k1[55], k1[51], k1[53], k1[28],
  k1[41], k1[50], k1[35], k1[46], k1[33], k1[37], k1[44], k1[52],
  k1[30], k1[48], k1[40], k1[49], k1[29], k1[36], k1[43], k1[54],
  k1[15], k1[ 4], k1[25], k1[19], k1[ 9], k1[ 1], k1[26], k1[16],
  k1[ 5], k1[11], k1[23], k1[ 8], k1[12], k1[ 7], k1[17], k1[ 0],
  k1[22], k1[ 3], k1[10], k1[14], k1[ 6], k1[20], k1[27], k1[24]};

assign kout2 = {
  k2[42], k2[39], k2[45], k2[32], k2[55], k2[51], k2[53], k2[28],
  k2[41], k2[50], k2[35], k2[46], k2[33], k2[37], k2[44], k2[52],
  k2[30], k2[48], k2[40], k2[49], k2[29], k2[36], k2[43], k2[54],
  k2[15], k2[ 4], k2[25], k2[19], k2[ 9], k2[ 1], k2[26], k2[16],
  k2[ 5], k2[11], k2[23], k2[ 8], k2[12], k2[ 7], k2[17], k2[ 0],
  k2[22], k2[ 3], k2[10], k2[14], k2[ 6], k2[20], k2[27], k2[24]};

assign kout3 = {
  k3[42], k3[39], k3[45], k3[32], k3[55], k3[51], k3[53], k3[28],
  k3[41], k3[50], k3[35], k3[46], k3[33], k3[37], k3[44], k3[52],
  k3[30], k3[48], k3[40], k3[49], k3[29], k3[36], k3[43], k3[54],
  k3[15], k3[ 4], k3[25], k3[19], k3[ 9], k3[ 1], k3[26], k3[16],
  k3[ 5], k3[11], k3[23], k3[ 8], k3[12], k3[ 7], k3[17], k3[ 0],
  k3[22], k3[ 3], k3[10], k3[14], k3[ 6], k3[20], k3[27], k3[24]};

assign kout4 = {
  k4[42], k4[39], k4[45], k4[32], k4[55], k4[51], k4[53], k4[28],
  k4[41], k4[50], k4[35], k4[46], k4[33], k4[37], k4[44], k4[52],
  k4[30], k4[48], k4[40], k4[49], k4[29], k4[36], k4[43], k4[54],
  k4[15], k4[ 4], k4[25], k4[19], k4[ 9], k4[ 1], k4[26], k4[16],
  k4[ 5], k4[11], k4[23], k4[ 8], k4[12], k4[ 7], k4[17], k4[ 0],
  k4[22], k4[ 3], k4[10], k4[14], k4[ 6], k4[20], k4[27], k4[24]};

assign kout5 = {
  k5[42], k5[39], k5[45], k5[32], k5[55], k5[51], k5[53], k5[28],
  k5[41], k5[50], k5[35], k5[46], k5[33], k5[37], k5[44], k5[52],
  k5[30], k5[48], k5[40], k5[49], k5[29], k5[36], k5[43], k5[54],
  k5[15], k5[ 4], k5[25], k5[19], k5[ 9], k5[ 1], k5[26], k5[16],
  k5[ 5], k5[11], k5[23], k5[ 8], k5[12], k5[ 7], k5[17], k5[ 0],
  k5[22], k5[ 3], k5[10], k5[14], k5[ 6], k5[20], k5[27], k5[24]};

assign kout6 = {
  k6[42], k6[39], k6[45], k6[32], k6[55], k6[51], k6[53], k6[28],
  k6[41], k6[50], k6[35], k6[46], k6[33], k6[37], k6[44], k6[52],
  k6[30], k6[48], k6[40], k6[49], k6[29], k6[36], k6[43], k6[54],
  k6[15], k6[ 4], k6[25], k6[19], k6[ 9], k6[ 1], k6[26], k6[16],
  k6[ 5], k6[11], k6[23], k6[ 8], k6[12], k6[ 7], k6[17], k6[ 0],
  k6[22], k6[ 3], k6[10], k6[14], k6[ 6], k6[20], k6[27], k6[24]};

assign kout7 = {
  k7[42], k7[39], k7[45], k7[32], k7[55], k7[51], k7[53], k7[28],
  k7[41], k7[50], k7[35], k7[46], k7[33], k7[37], k7[44], k7[52],
  k7[30], k7[48], k7[40], k7[49], k7[29], k7[36], k7[43], k7[54],
  k7[15], k7[ 4], k7[25], k7[19], k7[ 9], k7[ 1], k7[26], k7[16],
  k7[ 5], k7[11], k7[23], k7[ 8], k7[12], k7[ 7], k7[17], k7[ 0],
```

```
    k7[22], k7[ 3], k7[10], k7[14], k7[ 6], k7[20], k7[27], k7[24]};

  assign kout8 = {
    k8[42], k8[39], k8[45], k8[32], k8[55], k8[51], k8[53], k8[28],
    k8[41], k8[50], k8[35], k8[46], k8[33], k8[37], k8[44], k8[52],
    k8[30], k8[48], k8[40], k8[49], k8[29], k8[36], k8[43], k8[54],
    k8[15], k8[ 4], k8[25], k8[19], k8[ 9], k8[ 1], k8[26], k8[16],
    k8[ 5], k8[11], k8[23], k8[ 8], k8[12], k8[ 7], k8[17], k8[ 0],
    k8[22], k8[ 3], k8[10], k8[14], k8[ 6], k8[20], k8[27], k8[24]};

  assign kout9 = {
    k9[42], k9[39], k9[45], k9[32], k9[55], k9[51], k9[53], k9[28],
    k9[41], k9[50], k9[35], k9[46], k9[33], k9[37], k9[44], k9[52],
    k9[30], k9[48], k9[40], k9[49], k9[29], k9[36], k9[43], k9[54],
    k9[15], k9[ 4], k9[25], k9[19], k9[ 9], k9[ 1], k9[26], k9[16],
    k9[ 5], k9[11], k9[23], k9[ 8], k9[12], k9[ 7], k9[17], k9[ 0],
    k9[22], k9[ 3], k9[10], k9[14], k9[ 6], k9[20], k9[27], k9[24]};

  assign kout10 = {
    k10[42], k10[39], k10[45], k10[32], k10[55], k10[51], k10[53], k10[28],
    k10[41], k10[50], k10[35], k10[46], k10[33], k10[37], k10[44], k10[52],
    k10[30], k10[48], k10[40], k10[49], k10[29], k10[36], k10[43], k10[54],
    k10[15], k10[ 4], k10[25], k10[19], k10[ 9], k10[ 1], k10[26], k10[16],
    k10[ 5], k10[11], k10[23], k10[ 8], k10[12], k10[ 7], k10[17], k10[ 0],
    k10[22], k10[ 3], k10[10], k10[14], k10[ 6], k10[20], k10[27], k10[24]};

  assign kout11 = {
    k11[42], k11[39], k11[45], k11[32], k11[55], k11[51], k11[53], k11[28],
    k11[41], k11[50], k11[35], k11[46], k11[33], k11[37], k11[44], k11[52],
    k11[30], k11[48], k11[40], k11[49], k11[29], k11[36], k11[43], k11[54],
    k11[15], k11[ 4], k11[25], k11[19], k11[ 9], k11[ 1], k11[26], k11[16],
    k11[ 5], k11[11], k11[23], k11[ 8], k11[12], k11[ 7], k11[17], k11[ 0],
    k11[22], k11[ 3], k11[10], k11[14], k11[ 6], k11[20], k11[27], k11[24]};

  assign kout12 = {
    k12[42], k12[39], k12[45], k12[32], k12[55], k12[51], k12[53], k12[28],
    k12[41], k12[50], k12[35], k12[46], k12[33], k12[37], k12[44], k12[52],
    k12[30], k12[48], k12[40], k12[49], k12[29], k12[36], k12[43], k12[54],
    k12[15], k12[ 4], k12[25], k12[19], k12[ 9], k12[ 1], k12[26], k12[16],
    k12[ 5], k12[11], k12[23], k12[ 8], k12[12], k12[ 7], k12[17], k12[ 0],
    k12[22], k12[ 3], k12[10], k12[14], k12[ 6], k12[20], k12[27], k12[24]};

  assign kout13 = {
    k13[42], k13[39], k13[45], k13[32], k13[55], k13[51], k13[53], k13[28],
    k13[41], k13[50], k13[35], k13[46], k13[33], k13[37], k13[44], k13[52],
    k13[30], k13[48], k13[40], k13[49], k13[29], k13[36], k13[43], k13[54],
    k13[15], k13[ 4], k13[25], k13[19], k13[ 9], k13[ 1], k13[26], k13[16],
    k13[ 5], k13[11], k13[23], k13[ 8], k13[12], k13[ 7], k13[17], k13[ 0],
    k13[22], k13[ 3], k13[10], k13[14], k13[ 6], k13[20], k13[27], k13[24]};

  assign kout14 = {
    k14[42], k14[39], k14[45], k14[32], k14[55], k14[51], k14[53], k14[28],
    k14[41], k14[50], k14[35], k14[46], k14[33], k14[37], k14[44], k14[52],
    k14[30], k14[48], k14[40], k14[49], k14[29], k14[36], k14[43], k14[54],
    k14[15], k14[ 4], k14[25], k14[19], k14[ 9], k14[ 1], k14[26], k14[16],
    k14[ 5], k14[11], k14[23], k14[ 8], k14[12], k14[ 7], k14[17], k14[ 0],
    k14[22], k14[ 3], k14[10], k14[14], k14[ 6], k14[20], k14[27], k14[24]};

  assign kout15 = {
    k15[42], k15[39], k15[45], k15[32], k15[55], k15[51], k15[53], k15[28],
    k15[41], k15[50], k15[35], k15[46], k15[33], k15[37], k15[44], k15[52],
    k15[30], k15[48], k15[40], k15[49], k15[29], k15[36], k15[43], k15[54],
    k15[15], k15[ 4], k15[25], k15[19], k15[ 9], k15[ 1], k15[26], k15[16],
    k15[ 5], k15[11], k15[23], k15[ 8], k15[12], k15[ 7], k15[17], k15[ 0],
    k15[22], k15[ 3], k15[10], k15[14], k15[ 6], k15[20], k15[27], k15[24]};

  assign kout16 = {
    k16[42], k16[39], k16[45], k16[32], k16[55], k16[51], k16[53], k16[28],
    k16[41], k16[50], k16[35], k16[46], k16[33], k16[37], k16[44], k16[52],
    k16[30], k16[48], k16[40], k16[49], k16[29], k16[36], k16[43], k16[54],
    k16[15], k16[ 4], k16[25], k16[19], k16[ 9], k16[ 1], k16[26], k16[16],
    k16[ 5], k16[11], k16[23], k16[ 8], k16[12], k16[ 7], k16[17], k16[ 0],
    k16[22], k16[ 3], k16[10], k16[14], k16[ 6], k16[20], k16[27], k16[24]};

  // Sauvegarde dans le register interne
  assign KEYSC = {
    kout16, kout15, kout14, kout13, kout12, kout11, kout10, kout9,
    kout8,  kout7,  kout6,  kout5,  kout4,  kout3,  kout2,  kout1};
}
```

V.3.2 Code C++ utilisant les instructions TIE

```
// Chiffrement d'un bloc de 64 bits de données (L et R) avec une nouvelle clé
void tie_des_encrypt_key(unsigned int &L, unsigned int &R,
           const unsigned int key_msb, const unsigned int key_lsb)
{
    unsigned int t1,t2;
    t1 = L;
    t2 = R;
    L = DES_IPa(t1,t2);
    R = DES_IPb(t1,t2);
    DES_KEYGEN(key_msb,key_lsb);

    L = L^DES_FCT(R,0);
    R = R^DES_FCT(L,1);
    L = L^DES_FCT(R,2);
    R = R^DES_FCT(L,3);
    L = L^DES_FCT(R,4);
    R = R^DES_FCT(L,5);
    L = L^DES_FCT(R,6);
    R = R^DES_FCT(L,7);
    L = L^DES_FCT(R,8);
    R = R^DES_FCT(L,9);
    L = L^DES_FCT(R,10);
    R = R^DES_FCT(L,11);
    L = L^DES_FCT(R,12);
    R = R^DES_FCT(L,13);
    L = L^DES_FCT(R,14);
    R = R^DES_FCT(L,15);
    t1 = R;
    t2 = L;
    L = DES_REVIPa(t1,t2);
    R = DES_REVIPb(t1,t2);
}
```

V.3.3 Code utilisé pour valider les instructions TIE

```
void tie_des_testencrypt()
{
    /*
      Valeurs définies dans la table 4 de la publication spéciale
      800-17 du NIST
      Table 4 :
      Values To Be Used for the Substitution Table Known Answer Test
          KEY                PT                CT
      7CA110454A1A6E57   01A1D6D039776742   690F5B0D9A26939B
      0131D9619DC1376E   5CD54CA83DEF57DA   7A389D10354BD271
      07A1133E4A0B2686   0248D43806F67172   868EBB51CAB4599A
      3849674C2602319E   51454B582DDF440A   7178876E01F19B2A
      04B915BA43FEB5B6   42FD443059577FA2   AF37FB421F8C4095
      0113B970FD34F2CE   059B5E0851CF143A   86A560F10EC6D85B
      0170F175468FB5E6   0756D8E0774761D2   0CD3DA020021DC09
      43297FAD38E373FE   762514B829BF486A   EA676B2CB7DB2B7A
      07A7137045DA2A16   3BDD119049372802   DFD64A815CAF1A0F
      04689104C2FD3B2F   26955F6835AF609A   5C513C9C4886C088
      37D06BB516CB7546   164D5E404F275232   0A2AEEAE3FF4AB77
      1F08260D1AC2465E   6B056E18759F5CCA   EF1BF03E5DFA575A
      584023641ABA6176   004BD6EF09176062   88BF0DB6D70DEE56
      025816164629B007   480D39006EE762F2   A1F9915541020B56
```

```
      49793EBC79B3258F   437540C8698F3CFA   6FBF1CAFCFFD0556
      4FB05E1515AB73A7   072D43A077075292   2F22E49BAB7CA1AC
      49E95D6D4CA229BF   02FE55778117F12A   5A6B612CC26CCE4A
      018310DC409B26D6   1D9D5C5018F728C2   5F4C038ED12B2E41
      1C587F1C13924FEF   305532286D6F295A   63FAC0D034D9F793
*/

    // Clés, les deux dernières valeurs proviennent de l'annexe A de 800-17
    unsigned int k[40] =
    { 0x7CA11045, 0x4A1A6E57, 0x0131D961, 0x9DC1376E, 0x07A1133E, 0x4A0B2686,
      0x3849674C, 0x2602319E, 0x04B915BA, 0x43FEB5B6, 0x0113B970, 0xFD34F2CE,
      0x0170F175, 0x468FB5E6, 0x43297FAD, 0x38E373FE, 0x07A71370, 0x45DA2A16,
      0x04689104, 0xC2FD3B2F, 0x37D06BB5, 0x16CB7546, 0x1F08260D, 0x1AC2465E,
      0x58402364, 0x1ABA6176, 0x02581616, 0x4629B007, 0x49793EBC, 0x79B3258F,
      0x4FB05E15, 0x15AB73A7, 0x49E95D6D, 0x4CA229BF, 0x018310DC, 0x409B26D6,
      0x1C587F1C, 0x13924FEF, 0x10316E02, 0x8C8F3B4A };

    // Texte à chiffrer
    unsigned int pt[40] =
    { 0x01A1D6D0, 0x39776742, 0x5CD54CA8, 0x3DEF57DA, 0x0248D438, 0x06F67172,
      0x51454B58, 0x2DDF440A, 0x42FD4430, 0x59577FA2, 0x059B5E08, 0x51CF143A,
      0x0756D8E0, 0x774761D2, 0x762514B8, 0x29BF486A, 0x3BDD1190, 0x49372802,
      0x26955F68, 0x35AF609A, 0x164D5E40, 0x4F275232, 0x6B056E18, 0x759F5CCA,
      0x004BD6EF, 0x09176062, 0x480D3900, 0x6EE762F2, 0x437540C8, 0x698F3CFA,
      0x072D43A0, 0x77075292, 0x02FE5577, 0x8117F12A, 0x1D9D5C50, 0x18F728C2,
      0x30553228, 0x6D6F295A, 0x00000000, 0x00000000 };

    // Résultat attendu
    unsigned int ct[40] =
    { 0x690F5B0D, 0x9A26939B, 0x7A389D10, 0x354BD271, 0x868EBB51, 0xCAB4599A,
      0x7178876E, 0x01F19B2A, 0xAF37FB42, 0x1F8C4095, 0x86A560F1, 0x0EC6D85B,
      0x0CD3DA02, 0x0021DC09, 0xEA676B2C, 0xB7DB2B7A, 0xDFD64A81, 0x5CAF1A0F,
      0x5C513C9C, 0x4886C088, 0x0A2AEEAE, 0x3FF4AB77, 0xEF1BF03E, 0x5DFA575A,
      0x88BF0DB6, 0xD70DEE56, 0xA1F99155, 0x41020B56, 0x6FBF1CAF, 0xCFFD0556,
      0x2F22E49B, 0xAB7CA1AC, 0x5A6B612C, 0xC26CCE4A, 0x5F4C038E, 0xD12B2E41,
      0x63FAC0D0, 0x34D9F793, 0x82DCBAFB, 0xDEAB6602 };

    int i,j;
    bool passed = true;
    printf("tie_des_testencrypt() : Test DES algo... ");
    for (i=0; i<40; i+=2) {
        j = i+1;
        tie_des_encrypt_key(pt[i],pt[j],k[i],k[j]);
        if ( (pt[i] != ct[i]) || (pt[j] != ct[j]) ) {
            printf("Test failed ! (i=%d)\n",i);
            printf("\tResult=%08X%08X Expected=%08X%08X\n",
                   pt[i],pt[j],ct[i],ct[j]);
            passed = false;
        }
    }
    if (passed)
        printf("test completed with success !\n");
    else
        printf("test completed but some of them failed\n");
}
```

ANNEXE VI

DÉTAILS DES GAINS OBTENUS LORS DES OPTIMISATIONS

VI.1 Application IPv4

Le tableau VI.1 résume les accélérations obtenues pour l'application IPv4. Les gains sont séparés selon quatre types d'optimisations qui suivent la méthodologie proposée à la section 2.4. Le tableau VI.1 donne aussi le détail de gains lorsque des paquets de taille minimale et des paquets de taille variable sont employés. De plus, les gains présentés sont cumulatifs puisque les optimisations réalisées précédemment ne sont pas retirés lorsque l'on passe à l'étape subséquente de la méthodologie.

TABLEAU VI.1 Gains obtenus pour l'application IPv4

Catégories	Gains pour chaque type d'optimisation							
	Modification du logiciel		Configuration du processeur		Instructions TIE		Coprocesseur	
	Min.	Var.	Min.	Var.	Min.	Var.	Min.	Var.
Window Overflow et Underflow	0.94	0.95	1.71	1.78	1.65	1.76	1.89	2.03
Fonction Push	1.21	1.11	1.18	1.10	1.19	1.04	1.31	1.40
Gestion de la mémoire	1.00	1.00	1.07	0.99	1.00	0.99	2.74	12.84
Paquets IN / OUT	1.03	1.13	1.08	1.13	1.15	1.18	1.08	1.17
Vérification de l'en-tête IP	1.34	1.09	1.42	0.82	1.37	1.00	1.40	0.99
Calcul du checksum	1.05	1.12	1.04	0.98	2.00	1.77	2.36	2.18
Autres	1.11	0.95	1.13	0.67	1.47	1.06	2.45	1.53

Les gains présentés ici sont pour l'application Click complète, cette application contient beaucoup de fonctions qui n'ont pas étés optimisées et c'est pourquoi un bonne partie des gains du tableau VI.1 sont modestes. Dans le chapitre 3, les accélérations obtenus avec les instructions TIE étaient données uniquement pour la petite boucle optimisée et non pour l'application en entier. Pour l'application IPv4, l'optimisation la plus intéressante est de loin l'ajout d'un coprocesseur.

VI.2 Application IPSec

Le tableau VI.1 résume les accélérations obtenues pour l'application IPSec. Ici aussi, les gains sont cumulatifs et ils sont donnés pour des paquets de taille minimale et variable. Pour l'application IPSec, l'accélération offerte par les instructions TIE est beaucoup plus importante que dans le cas de l'application IPv4, cela est dû à l'ajout de l'algorithme de chiffrement DES.

TABLEAU VI.2 Gains obtenus pour l'application IPSec

Catégories	Gains pour chaque type d'optimisation							
	Modification du logiciel		Configuration du processeur		Instructions TIE		Coprocesseur	
	Min.	Var.	Min.	Var.	Min.	Var.	Min.	Var.
Chiffrement DES-CBC	1.03	1.01	1.03	1.00	6.59	7.96	7.00	7.93
Window Overflow et Underflow	1.01	0.97	1.93	1.91	1.95	1.94	2.87	2.93
Fonction Push	0.95	1.11	0.98	1.08	1.24	1.18	1.42	1.55
Gestion de la mémoire	1.09	1.09	0.95	1.04	1.15	1.10	5.22	9.83
Paquets IN / OUT	1.01	1.01	1.01	1.01	1.03	1.05	1.06	1.06
Vérification de l'en-tête IP	1.00	1.01	0.93	1.01	0.99	0.99	1.01	1.00
Calcul du checksum	1.02	0.97	1.03	1.02	1.90	1.85	2.16	2.22
Autres	1.09	0.78	1.38	1.06	1.12	1.13	2.26	2.99